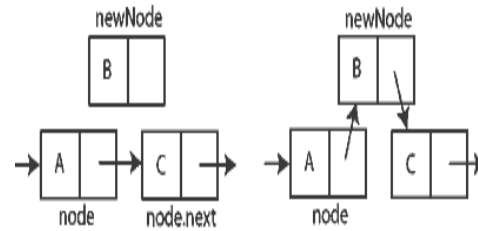
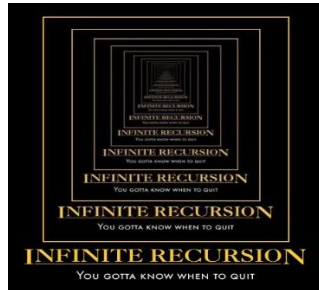


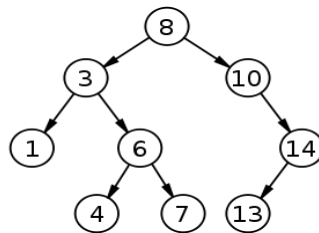


Universidad Autónoma del Estado de México

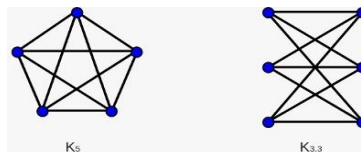
Centro Universitario UAEM Texcoco



## INGENIERÍA EN COMPUTACIÓN



## APUNTES



## ESTRUCTURA DE DATOS

2017A

Prof. Joel Ayala de la Vega

Centro Universitario UAEM Texcoco  
Av. Jardín Zumpango s/n. Fracc. El Tejocote  
C.P. 56259 Texcoco, Estado de México.  
Tels. (595) 9211216 - 9211247 - 9210368 - 9210493  
e-mail: cutex.uaem@gmail.com.

CUTex

## Contenido

INTRODUCCIÓN .....	5
PRESENTACIÓN.....	6
PRIMER CAPÍTULO: Categorías de Módulos.....	8
Resumen.....	8
1.2 Módulos .....	10
1.2.1 Procedimientos abstractos y librerías. ....	10
1.2.2 Agrupamiento común de datos.....	10
1.2.3 Objetos Abstractos .....	11
1.2.4 Tipos de Datos Abstractos.....	12
Ejercicios.....	14
SEGUNDO CAPITULO: &, Una constante tipo apuntador.....	15
Resumen .....	15
2.1 Variables automáticas.....	15
2.2 Arreglos de caracteres. ....	19
Ejercicios:.....	26
TERCER CAPITULO: *, Una variable tipo apuntador.....	27
Resumen .....	27
3.1 Variables tipo apuntador y operadores básicos.....	27
3.2 Variables tipo apuntador para cadenas de caracteres.....	30
3.3 Algunos errores en el uso de apuntadores tipo carácter.....	37
3.4 Arreglos de apuntadores. ....	39
3.5 Manejo de Estructuras (registros o tuplas).....	42
3.6 Apuntadores a apuntadores. ....	45
3.7 Manejo de matrices con apuntadores. ....	50
Ejercicios:.....	51
CUARTO CAPITULO: ALLOCATE. Asignar memoria en forma dinámica. ....	54
Resumen .....	54
4.1 Apuntadores y asignación de memoria. ....	54
4.2 Manejo de matrices en forma dinámica .....	61
4.3 Manejo de arreglos de registros en forma dinámica. ....	63
Ejercicios.....	65

<b>QUINTO CAPITULO: FUNCIONES.</b>	<b>66</b>
Resumen.	66
5.1 Paso de parámetros.	66
5.2 Paso de parámetros en vectores.	69
5.3 Paso de parámetros en matrices estáticas.	71
5.4 Paso de parámetros con matrices dinámicas.	74
5.5 Paso de parámetros en registros.	75
5.6 Apuntadores a funciones.	78
Ejercicios:	85
<b>SEXTO CAPITULO: LISTAS ENLAZADAS.</b>	<b>86</b>
Resumen.	86
6.1. Programación de listas enlazadas	86
6.2 Programación de una pila	86
6.3 Traducción de una expresión infija a postfija.	91
6.4 Programación de una lista simplemente enlazada ordenada.	95
6.6 Programación de una fila o cola.	104
6.7 Aplicaciones de listas enlazadas.	106
6.7.1 Aplicaciones en pilas	106
6.7.2 Aplicaciones de colas.	107
6.8 Ejercicios.	107
<b>SEPTIMO CAPÍTULO: Recursividad Directa.</b>	<b>109</b>
Resumen.	109
7.1 Definición de recursividad.	109
7.2 Clasificación de funciones recursivas.	110
7.3 Diseño de funciones recursivas	111
7.4 Ventajas e inconvenientes de la recursividad.	112
7.5 Cálculo de determinantes en forma recursiva	115
Ejercicios.	117
<b>OCTAVO CAPÍTULO: ÁRBOLES BINARIOS.</b>	<b>119</b>
Resumen	119
8.1 Introducción.	119
8.2 Árboles binarios de expresiones	120

8.3 Recorrido de un árbol binario.....	121
8.4.1 La inserción de un árbol binario de búsqueda .....	122
8.4.2 El borrado en un árbol binario de búsqueda. ....	122
8.4.3 Recorrido en un árbol binario de búsqueda .....	123
Ejercicios:.....	129
<b>NOVENO CAPÍTULO: GRAFOS.....</b>	<b>132</b>
Resumen.....	132
9.1 Introducción .....	132
9.2 Definición de grafo. ....	133
9.3 Árbol de expansión mínima (Minimun Spanning Tree).....	136
9.4 La ruta más corta a partir de un origen (Single Source Shortest Paths).....	139
Ejercicios.....	145
<b>Bibliografía .....</b>	<b>146</b>

## INTRODUCCIÓN

Para un estudiante en Ingeniería en Computación es fundamental el estudio de las estructuras de datos ya que se requieren en Unidades de Aprendizaje donde se desarrolla software que se basa en tales estructuras. Como ejemplo de esto se tiene a la Unidad de Aprendizaje “Métodos Numéricos” donde se requiere la utilización de vectores y matrices para poder resolver derivadas, integrales numéricas y sistemas de ecuaciones lineales, la Unidad de aprendizaje “Programación Avanzada” donde se requiere el uso de matrices con el manejo de memoria dinámica para trabajar con grafos, la Unidad de Aprendizaje “Compiladores” donde se requiere el uso de estructuras enlazadas en forma dinámica para poder manejar un árbol sintáctico (parser tree), etc.

Las estructuras pueden ser estáticas (dadas por el propio compilador) o dinámicas (las que crea el propio desarrollador de software). La asignación de memoria en la estructura estática generalmente se realiza en el momento de la compilación. En una estructura dinámicas se puede requerir el manejo dinámico de la memoria (La solicitud de la memoria en el momento de la ejecución). Viendo la importancia de comprender plenamente el manejo de la memoria en forma dinámica se escribieron 3 capítulos de estos apuntes (El primer capítulo explica el operador “&” y cómo hacer referencia a un área de memoria. El segundo capítulo con el operador “\*”, como apuntar en forma específica a un área de memoria. Y el tercer capítulo con las funciones malloc, calloc y realloc para crear espacio de memoria en forma dinámica). Además de explicar el truco que se utiliza para simular el pase de parámetros por referencia. De esta forma se espera que el alumno tenga una conciencia plena del uso dinámico de la memoria para variables primitivas, arreglos y registros.

Conociendo el material presentado en los capítulos sobre memoria dinámica, se puede comprender mejor la forma que se trabajan estructuras dinámicas clásicas como son las listas enlazadas, doblemente enlazadas, Por lo que un capítulo está dedicado a este tipo de estructuras y otro capítulo se dedica al estudio de árboles binarios.

El último capítulo corresponde al manejo de grafos tanto dirigidos como no dirigidos. Para grafos no dirigidos se escogió el algoritmo del árbol de expansión mínima y para grafos dirigidos se escogió el algoritmo de la ruta más corta a partir de un origen. Ambos algoritmos se programaron con matrices creadas en forma dinámica.

La programación del material aquí presentado es de propia autoría. En los capítulos que sea requerido se indica la bibliografía donde se tomó el pseudocódigo para la programación del algoritmo indicado.

## PRESENTACIÓN.

Este escrito fue hecho con la intención de apoyar a los estudiantes que cursan la Unidad de Aprendizaje “**Estructura de Datos**”. El plan de estudios aprobado en el 2013 comprenden los siguientes temas:

Unidad de competencias I.

- *Estructura de Datos*: Definición y tipos.
- *Abstracción*: Definición TAD.
- *Variables dinámicas*: Apuntadores. Operaciones Básicas.

Unidad de competencias II

- *Pilas*: Representación. Operaciones (Inserción, Eliminación, Pila Llena, Pila Vacía). Tratamiento de expresiones aritméticas (Notación Infija, Prefija, Postfija). Aplicaciones.
- *Colas*: Representación. Operaciones (Inserción, Eliminación, Cola Llena, Cola Vacía). Cola circular. Aplicaciones.
- *Listas*: Representación. Operaciones (Inserción, Eliminación, Recorrido, Búsqueda). Listas doblemente ligadas. Aplicaciones.

Unidad de competencias III.

- *Recursividad Directa*: Definición. Funcionamiento.
- *Árboles*: Usos. Características.
- *Árboles Binarios*: Representación. Operaciones. Recorrido (Preorden, Orden, Posorden).
- *Árboles Binarios de Expresiones*: Características. Evaluación de expresiones aritméticas.
- *Árboles Binarios de Búsqueda*: Características. Operaciones (Inserción, Eliminación, Búsqueda).

Unidad de competencias IV.

- *Grafos*: Características. Tipos. Representación y construcción. Operaciones
- *Grafos Dirigidos*: Algoritmos para la obtención del camino más corto.
- *Grafos No dirigidos*: Algoritmos para la obtención de costo mínimo.

Para lograr esto, se escribieron los siguientes capítulos:

**Primer capítulo:** Se define que es “estructura de datos” y los “tipos de datos” que se manejan. También comenta los tipos de módulos existentes, entre los más importantes son los “Tipos de Datos Abstractos” (TDA).

**Del segundo al quinto capítulo:** Se explica el manejo de la memoria dinámica y su uso con variables primitivas como los enteros, flotantes, char. También muestra el uso de la memoria dinámica con arreglos, matrices y registros. Además se explica el pase de parámetros por valor y por referencia.

Los capítulos uno al cinco cubren **plenamente la Unidad de Competencias I.**

**El sexto capítulo:** Se dedica al estudio de listas enlazadas, principalmente las listas simplemente enlazadas como las colas, filas, listas y listas doblemente enlazadas. De esta forma, se proporciona el material necesario para **cubrir plenamente la Unidad de Competencias II.**

**El séptimo capítulo:** Se realiza un estudio de la recursividad.

**El octavo capítulo:** Se exponen los árboles binarios, su recorrido, evaluación de expresiones y árboles binarios de búsqueda.

Por lo que, entre los capítulos siete y ocho se **cubre plenamente la Unidad de Competencias III.**

**El noveno capítulo:** Se explica lo que son grafos y se dan dos ejemplos clásicos de ellos: El árbol de expansión mínima y la ruta más corta desde un origen. Con este material se **cubre la Unidad de Competencias IV.**

## PRIMER CAPÍTULO: Categorías de Módulos.

### Resumen.

Este capítulo forma parte de la primera unidad de competencia en la parte de “Estructuras de datos: Definición, Tipos” y “Abstracción: Definición. TAD”. La idea de Estructura de datos se tomó principalmente del Cairó (Cairó/Gardati, 2000). Ghezzi (Ghezzi, Jazayeri, & Mandrioli, 1991) da una explicación muy interesante de lo que es TAD, por lo que se recomienda, si es posible, leer el capítulo cuatro de tal libro donde se explica a fondo el diseño modular y conceptos fundamentales de TAD

### 1.1 Definición de Estructura de Datos.

En programación, una **estructura de datos** es una forma particular de organizar datos en una computadora para que pueda ser utilizado de manera eficiente.

Diferentes tipos de estructuras de datos son adecuados para diferentes tipos de aplicaciones, y algunos son altamente especializados para tareas específicas.

Las estructuras de datos son un medio para manejar grandes cantidades de datos de manera eficiente para usos tales como grandes bases de datos y servicios de indización de Internet. Por lo general, las estructuras de datos eficientes son la clave para diseñar algoritmos eficientes. Algunos métodos formales de diseño y lenguajes de programación destacan las estructuras de datos, en lugar de los algoritmos, como el factor clave de organización en el diseño de software.

Las estructuras de datos se basan generalmente en la capacidad de un ordenador para recuperar y almacenar datos en cualquier lugar de su memoria.

Existen numerosos tipos de estructuras de datos, generalmente construidas sobre otras más simples:

- Un arreglo es una serie de elementos en un orden específico, por lo general todos del mismo tipo (si bien los elementos pueden ser de casi cualquier tipo). Se accede a los elementos utilizando un entero como índice para especificar el elemento que se requiere. Las implementaciones típicas asignan palabras de memoria contiguas a los elementos de los arreglos (aunque no siempre es el caso). Los arreglos pueden cambiar de tamaño o tener una longitud fija.
- Un arreglo asociativo (también llamado *diccionario* o *mapa*) es una variante más flexible que una matriz, en la que se puede añadir y eliminar libremente pares nombre-valor. Una tabla de hash es una implementación usual de un arreglo asociativo.
- Un registro (también llamado *tupla* o *estructura*) es una estructura de datos agregados. Un registro es un valor que contiene otros valores, típicamente en un número fijo y la



secuencia y por lo general un índice por nombres. Los elementos de los registros generalmente son llamados *campos*.

- Una unión es una estructura de datos que especifica cuál de una serie de tipos de datos permitidos podrá ser almacenada en sus instancias, por ejemplo *flotante* o *entero largo*. En contraste con un registro, que se podría definir para contener un *flotante* y un *entero largo*, en una unión, sólo hay un valor a la vez. Se asigna suficiente espacio para contener el tipo de datos de cualquiera de los miembros.
- Un tipo variante (también llamado *registro variante* o *unión discriminada*) contiene un campo adicional que indica su tipo actual.
- Un conjunto es un tipo de datos abstracto que puede almacenar valores específicos, sin orden particular y sin valores duplicados.
- Un Multiconjunto es un tipo de datos abstracto que puede almacenar valores específicos, sin orden particular. A diferencia de los conjuntos, los multicunjuntos admiten repeticiones.
- Una lista es una colección de elementos llamados nodos. El orden entre nodos se establece por medio de apuntadores usando memoria dinámica, es decir, direcciones o referencias a otros nodos.
- Un grafo es una estructura de datos conectada compuesta por nodos. Cada nodo contiene un valor y una o más referencias a otros nodos. Los grafos pueden utilizarse para representar redes, dado que los nodos pueden referenciarse entre ellos. Las conexiones entre nodos pueden tener dirección, es decir un nodo de partida y uno de llegada.
- Un árbol es un caso particular de grafo dirigido en el que no se admiten ciclos y existe un camino desde un nodo llamado raíz hasta cada uno de los otros nodos. Una colección de árboles es llamada un bosque.
- Una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan como representación abstracta de conceptos, incluyen campos como los registros y operaciones que pueden consultar el valor de los campos o cambiar sus valores.

La mayoría de los lenguajes ensambladores y algunos lenguajes de bajo nivel carecen de soporte de estructuras de datos. En cambio, muchos lenguajes de alto nivel y algunos lenguajes ensambladores de alto nivel, tales como MASM, tienen algún tipo de soporte incorporado para ciertas estructuras de datos, tales como los registros y arreglos. Por ejemplo, el lenguaje C soporta variables simples estáticas, dinámicas y registros, respectivamente, además de arreglos y matrices multidimensionales. La mayoría de los lenguajes de programación disponen de algún tipo de biblioteca o mecanismo en el uso de

estructuras de los programas. Los lenguajes modernos por lo general vienen con bibliotecas estándar que implementan las estructuras de datos más comunes. Ejemplos de ello son la biblioteca `stdlib` de C, las colecciones de Java y las librerías .NET de Microsoft.

En programación, una estructura de datos puede ser declarada inicialmente escribiendo una palabra reservada, luego un identificador para la estructura y un nombre para cada uno de sus miembros, sin olvidar los tipos de datos que estos representan. Generalmente, cada miembro se separa con algún tipo de operador, carácter o palabra reservada.

## 1.2 Módulos

Aunque el diseñador pueda diseñar un módulo que exporte cualquier combinación de recursos (variables, tipos, procedimientos y funciones, etiquetas, etc.), la mayoría de los módulos pueden ser clasificados dentro de ciertas categorías estándar. Dicha categorización es útil porque, provee un esquema uniforme de clasificación para documentación y posiblemente, para la recuperación a partir de una biblioteca de componentes. También, utilizando un limitado conjunto de categorías hace al diseño más uniforme y estándar (piezas estándar+ son el signo de la madurez de una disciplina en ingeniería). Categorización de módulos es un paso hacia el desarrollo de componentes estándar en Ingeniería de Software.

Se discutirán tres categorías estándar: **abstracción procedural, librerías y agrupamiento común de datos** (common pools of data). Y dos categorías más generales y abstractas: **objetos abstractos y tipos de datos abstractos**.

Un tipo de modulo útil provee sólo un procedimiento o función, correspondiendo a alguna operación abstracta. Dichos módulos implementan una *abstracción procedural* y son usados para encapsular un algoritmo. Ejemplos típicos son los módulos de clasificación, módulos de la transformada rápida de Fourier, módulos que transforman un lenguaje a otro.

### 1.2.1 Procedimientos abstractos y librerías.

Un módulo puede contener un *grupo* de procedimientos abstractos relacionados. Un típico ejemplo se observa en las librerías de matemáticas. Dichas librerías proveen soluciones a los problemas más comunes en cálculo como la computación de límites, derivadas e integrales.

### 1.2.2 Agrupamiento común de datos

Otro tipo común de módulos provee una agrupación común de datos. Una vez que se reconoce la necesidad de compartir datos entre varios módulos, se pueden agrupar dichos datos en una agrupación común que es importada para todos los módulos de clientes.

Todos los módulos de cliente pueden manipular los datos en forma directa, de acuerdo a la estructura usada para representar los datos, los cuales son visibles a ellos.

En general, una agrupación común de datos es un módulo de bajo nivel. Tal módulo no provee ninguna forma de abstracción: todos los detalles de los datos son visibles y manipulados por los clientes. La habilidad de agrupar datos compartidos en un block común sólo provee una ayuda limitada en términos de lectura y modificabilidad.

Establecer grupos comunes de datos es fácilmente implementable en lenguajes de programación convencionales. Por ejemplo, se puede realizar en FORTRAN por medio del constructor COMMON, o en C utilizando variables estáticas.

Lo que interesa es tener módulos con mayor capacidad de abstracción que pueden *ocultar* estructuras particulares de datos como secretos del módulo. Por ejemplo, el módulo de la tabla de símbolos usada en un intérprete o en un compilador. Este módulo es un ejemplo de módulos que empaquetan tanto a los datos como a las rutinas.

### 1.2.3 Objetos Abstractos

El 17% de costos que involucran el desarrollo del mantenimiento de software se deben a los cambios de la representación de los datos. Por lo que es muy importante enmarcar que la **encapsulación** permite ocultar los detalles de la representación de los datos y protege a los clientes de los cambios en ellos.

Un módulo formado por una tabla de símbolos representa el típico módulo que oculta una estructura de datos como un secreto y exporta rutinas que pueden ser usadas para tener acceso a la estructura de datos oculta. La estructura de datos podría cambiar, todo lo que se necesita hacer es cambiar los algoritmos que implementan el acceso a las rutinas, pero los módulos del cliente continúan usando los mismos tipos de llamadas para ejecutar los accesos requeridos y de esta forma no se requieren cambios al exterior, por lo que las interfaces para el acceso de rutinas permanecen idénticas.

Para estas interfaces, estos tipos de módulos se observan como librerías. Pero tienen propiedades especiales, por ejemplo, las librerías de matemáticas tienen una estructura permanente de datos encapsulada en su implementación que es visible en rutinas que son internas al módulo, pero son ocultas a los módulos del cliente. En el ejemplo de la tabla de símbolos, la estructura de datos se utiliza para almacenar las entradas a medida que se insertan en la tabla.

La estructura de datos oculta provee esos módulos con un estado. En realidad, como una consecuencia de las llamadas a las rutinas exportadas por el modulo, los valores guardados en la estructura de datos puede cambiar de una llamada a otra; por lo tanto, el resultado

provisto por dos llamadas con exactamente los mismos parámetros puede ser diferente de un tiempo a otro. Este comportamiento es diferente para el caso de un grupo de procedimientos o funciones que constituyen una librería, ya que la librería no tiene un estado en particular: Dos llamadas sucesivas con los mismos parámetros siempre tendrán los mismos resultados.

La diferencia entre un módulo con un estado y módulos tipo librería no se muestra a través de la sintaxis de la interface. En ambos casos, el módulo exporta sólo un conjunto de subrutinas. Sin embargo, se tiene una distinción entre estos dos tipos de módulos. Módulos que exhiben un estado se llamarán *objetos abstractos*.

#### 1.2.4 Tipos de Datos Abstractos.

Un Tipo de Dato Abstracto (ADT por Abstract Data Type) es un módulo que exporta un tipo, junto con sus operaciones que deben ser invocadas para poder tener acceso y manipular objetos de tal tipo; oculta la representación del tipo y el algoritmo usado en la operación.

ADT recibe tanta atención hoy en día, como tuvo durante las dos últimas décadas de programación estructurada. La ADT no sustituye la programación estructurada. Más bien, provee una formalización adicional que puede mejorar el proceso de desarrollo de programas.

Para comprender mejor ADT veamos en C un tipo incorporado **int**. Lo que viene a la mente es el concepto de un entero en matemáticas, pero en una computadora **int** no es precisamente un entero en matemáticas. En particular, por lo regular los **int** de computadora son muy limitados en tamaño. Por ejemplo, en una máquina de 32 bits **int** pudiera estar limitado al rango desde -2 mil millones hasta +2 mil millones. Si el resultado de un cálculo cae fuera de este rango, ocurrirá un error y la máquina responderá de alguna manera (la respuesta depende del tipo de máquina). Los enteros matemáticos no tienen este problema. Por lo tanto, el concepto de un **int** en una computadora tiene un subconjunto de características del concepto del entero matemático del mundo real. Lo mismo funciona para **float**.

Inclusive **char** es también una aproximación. A menudo los valores tipo **char** son patrones de 8 bits, que no se parecen en nada a los caracteres que se suponen deben representar. En la mayor parte de las computadoras, los valores tipo **char** están bastante limitados en comparación con el rango de caracteres del mundo real. El conjunto de caracteres ASCII es totalmente inadecuado para representar el lenguaje japonés y el chino ya que sólo puede representar 255 caracteres, que requieren miles de caracteres (por lo que tuvo que aparecer UNICODE de 2 bytes para poder representar 65535 caracteres).

Los tipos de datos incorporados en un lenguaje de programación son sólo aproximaciones o modelos de conceptos y comportamientos del mundo real. Los tipos como **int**, **float** o **char** y otros son ejemplos de ADT. Son en esencia formas de representar conceptos del mundo real, a cierto nivel satisfactorio de precisión, dentro de un sistema de cómputo.

Un ADT de hecho captura dos conceptos, es decir una representación de datos, así como aquellas operaciones permitidas sobre dichos datos. Por ejemplo, en C, el concepto de **int** define la adición, substracción, multiplicación, división y módulo, pero queda indefinida la división entre cero; y estas operaciones permitidas se llevan a cabo de una forma que resulta dependiente de los parámetros de la máquina, como son el tamaño de la palabra fija del sistema de computación subyacente.

Ejemplo: ADT en arreglos.

Existen muchas operaciones que sería bueno ejecutar con arreglos, pero no están incorporadas en C. Con C el programador puede desarrollar un ADT. El ADT puede proveer nuevas capacidades como:

- Verificación del rango de los subíndices.
- Asignación de arreglo.
- Comparación de arreglo.
- Entrada/Salida de arreglo
- Conocer de su tamaño.
- Operaciones de algebra lineal.

Observaciones de Ingeniería de Software.

*El programador puede crear nuevos tipos. Estos nuevos tipos pueden ser diseñados para ser usados tan convenientemente como los tipos incorporados. A pesar de que mediante estos nuevos tipos el lenguaje es fácil de ampliar, el lenguaje base mismo no es modificable.*

Los nuevos ADT creados pueden ser propiedad de un individuo, de pequeños grupos o de empresas.

Ejemplo: ADT de fila.

Cada uno de nosotros tiene que hacer espera en vez de cuando en una fila (por ejemplo, la famosa fila de las tortillas). Los sistemas de cómputo utilizan de forma interna filas en espera, y se requieren escribir programas que simulen lo que la fila hace y es.

Una fila es un buen ejemplo de ADT. Los clientes ponen cosas en la fila utilizando una operación de *enqueue*, y los clientes obtienen estas cosas de regreso sobre demanda uno a

la vez utilizando la operación *dequeue*. Conceptualmente, una cola (fila) puede hacerse muy larga. Una cola real, por lo regular es finita. Los elementos de una cola son devueltos en un orden de primero en entrar, primero en salir (FIFO, *first in first out*)

La cola oculta una representación de datos internos que de alguna forma lleva control de los elementos actualmente esperando en fila, y ofrece un conjunto de operaciones a sus clientes, es decir *enqueue*, *dequeue*. Los clientes no tienen que preocuparse respecto a las puestas en práctica de la cola. Los clientes sólo desean que la cola funcione “como lo anunciado”. Cuando un cliente solicite el siguiente elemento de la parte frontal de la cola, la cola deberá quitar el elemento que ha estado por más tiempo en la cola, deberá ser el siguiente que, en la siguiente operación *dequeue*, será devuelto.

El ADT de la cola garantiza la integridad de su estructura interna de dato. Los clientes no pueden manipular en directo esta estructura de los datos. Solo el ADT de la cola tiene acceso a sus datos internos. Los clientes pueden realizar sólo operaciones permitidas, para ejecutarse sobre la representación de los datos; las operaciones no contempladas en la interfaz pública del ADT son rechazadas por el ADT de alguna forma apropiada. Esto podría significar la emisión de un mensaje de error, la terminación de la ejecución o sólo ignorar la solicitud d operación.

#### Ejercicios.

1. ¿Qué es una estructura de datos?
2. Comente los diferentes tipos de estructuras de datos.
3. Comente la categorización de módulos.
4. Comente qué es un objeto abstracto.
5. Comente qué es un tipo de dato abstracto.
6. Comente qué es un tipo de dato incorporado a un lenguaje y coloque dos ejemplos.
7. Explique un tipo de dato que no sea incorporado a un lenguaje, y operaciones permitidas.

## SEGUNDO CAPITULO: &, Una constante tipo apuntador.

### Resumen.

Este capítulo forma parte de la primera unidad de competencia en la parte de “Variables dinámicas: Apuntadores, Operaciones básicas” y muestra el primer el operador básico “&” que permite observar la dirección de memoria donde se localiza una variable. Es fundamental comprender el uso de tal operador ya que por medio de él se tiene una comprensión plena de la localización de elementos creados en forma estática como en forma dinámica en la memoria principal (la creación dinámica de variables se observa en el capítulo 4 de estos apuntes).

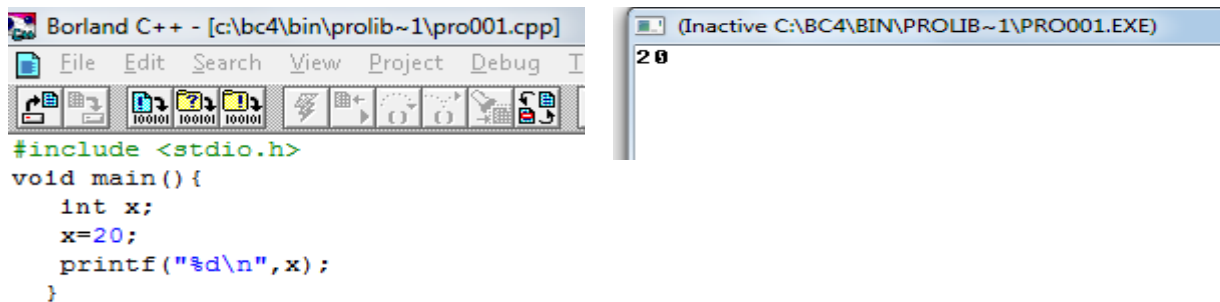
Los programas son de propia autoría y las ideas básicas se tomaron de los siguientes libros:  
(Deitel & Deitel, 1995)  
(Backman, 2012)

### 2.1 Variables automáticas.

Un apuntador en el lenguaje C, se entiende como una variable especial que guarda una dirección de memoria. Un apuntador no puede ser tratado como una variable estándar, por el adjetivo “especial”, esto indica que los apuntadores pueden ser tratados como variables estándar con algunos operadores, pero no pueden ser tratados en forma estándar con otros operadores. **Una programación descuidada (aunque técnicamente correcta) en el uso de apuntadores puede producir una mezcolanza casi imposible de entender.**

Las operaciones con apuntadores no son fáciles de entender, mientras no se comprendan los métodos utilizados en el lenguaje C y se fijen en la mente del programador. Estos métodos y reglas son manejadas por el compilador y son relativamente ocultas para el programador.

Iniciemos la discusión usando un programa simple como ejemplo:



Programa 2.1

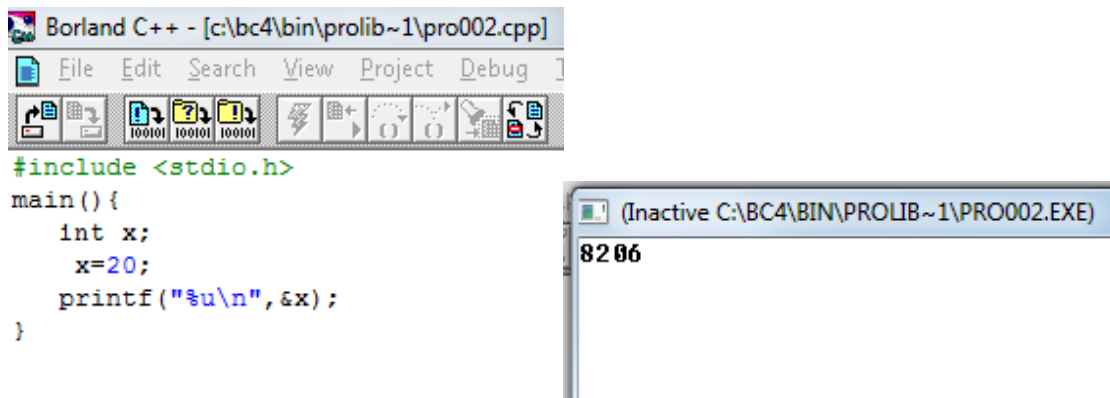
En el programa 2.1, - **x** - es declarada como variable estándar (auto) de tipo entero (int). Es asignado un valor y se muestra en la pantalla usando una función de escritura (printf( )). ¿Qué sucede cuando la variable es declarada? De manera sucinta describimos este proceso, “se reserva una localidad de la memoria con una extensión de dos bytes, para albergar un número entero”.

La línea de asignación indica que en la localidad se va a almacenar el valor 20. Siendo la declaración una variable entera con signo, indica que el rango de almacenamiento es desde 32,767 hasta -32,768 (El número de bytes depende del compilador, en este caso se están usando las especificaciones de Turbo C).

Cuando la función escribe (printf( )) es ejecutada, el valor colocado en - **x** - se desplegará en la pantalla como un entero.

Lo importante en esta declaración elemental es que se reservan dos bytes de almacenamiento “privado” para la variable - **x** -. Se puede acceder al espacio de memoria utilizando el operador “&” (también conocido como el “operador de direccionamiento”). En el programa 2.2 se muestra el valor de una dirección de memoria, al ejecutarse el programa puede aparecer un valor cualquiera que depende del compilador, del sistema operativo y tipo de máquina. Este valor es la dirección donde se guarda el valor de la variable - **x** -.





Programa 2.2

Siendo precedida la variable – **x** - con el ampersand, causa que la localidad de memoria, también llamada el “inicio de la dirección” de la variable, sea retornada en vez del valor entero guardado en la localidad de memoria. - **&x** - siempre hace referencia al inicio de la dirección de memoria dada para la variable – **x** -. **Nosotros no podemos cambiar esta dirección de memoria, es fija** (o constante).

En la función de impresión (printf()), la conversión “%u” se utiliza para desplegar un entero sin signo. Esto es necesario, ya que la dirección se localiza en la parte alta del segmento de la memoria pequeña (64 k bytes utilizados en Turbo C). El valor que se muestra en la pantalla depende de la configuración de la máquina, compilador, sistema operativo y uso de la memoria.

El punto importante a recordar es que la variable – **x** - es declarada una variable de tipo entero (int) y que - **&x** - es una constante (al menos durante la corrida del programa que estamos analizando). No puede ser cambiado y se encuentra atado a la variable **x** siendo igual al inicio de dirección del almacenamiento de la variable – **x** -. En otras palabras, no se puede colocar a la dirección de - **x** - (- **&x** -) otro valor. Por definición y diseño, siempre es igual al inicio de memoria asignado a **x**-. Por ejemplo La expresión:

**&x=9990;**

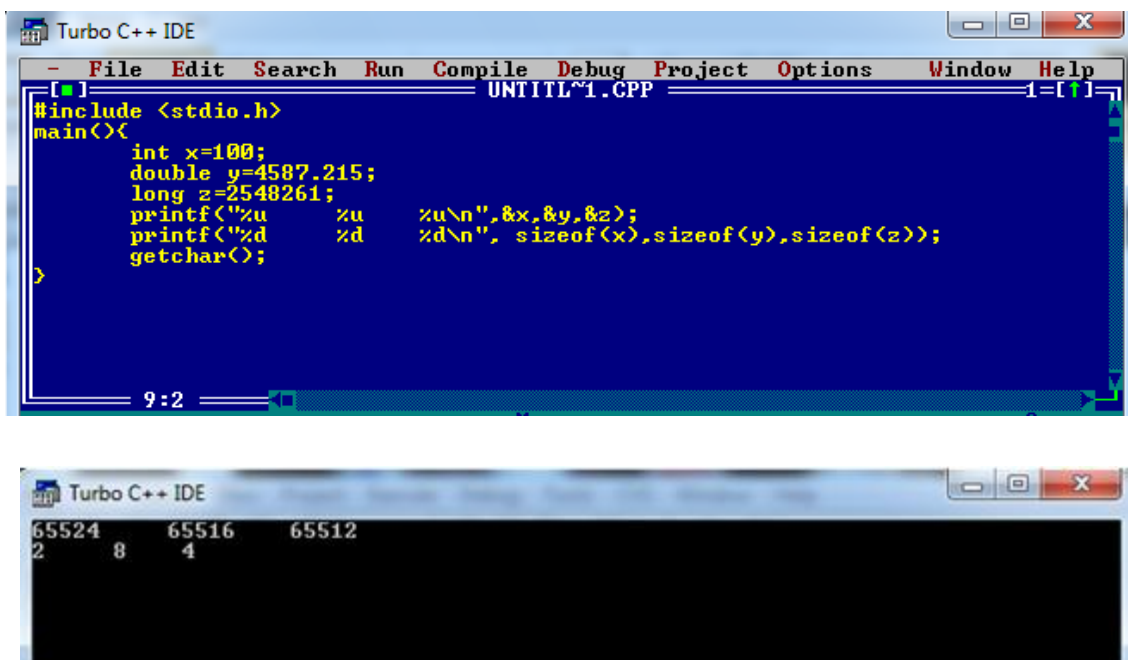
es completamente errónea, porque **&x** solo puede realizar una sola cosa, **retornar el valor que indica la dirección de almacenamiento de la variable – x -**. (Siendo la variable –**x**- entera, lo que retorna **&x** es la dirección del primer byte de almacenamiento).

En C, cuando se declara una variable sólo una operación básica se ejecuta, se asigna el espacio de memoria para cada variable declarada. Los bytes en esta localidad de memoria no son limpiados, en consecuencia contienen cualquier valor comúnmente denominada basura. Comúnmente se comete el error de pensar que el compilador inicializa la variable en cero (lenguajes como BASIC inicializan sus variables en cero). Si el programador asume erróneamente la inicialización de las variables, puede producir graves errores al momento de la ejecución del programa, esto es importante ya que sucede lo mismo con los

apuntadores. Todo apuntador no inicializado puede contener basura (Por ejemplo, para Turbo C será un valor entre 0 y 65535). Lo que hace un apuntador es contener o retornar una dirección de memoria. Lo que uno hace o deja de hacer con una dirección de memoria es responsabilidad del programador.

Cuando se declara una variable estándar, uno debe comprender que el espacio de almacenamiento es asignado automáticamente a la variable y se considera un almacenamiento “seguro”. Cuando un programa es cargado y ejecutado, existen diferentes porciones de la memoria que son reservados para diferentes cosas. Una porción de área se reserva para el manejo e interface con el sistema operativo, ésta es el área que permite al programa ser ejecutado en forma adecuada. El área de almacenamiento de variables no se localiza en tal área, ya que se puede producir una colisión, En consecuencia existe un área reservada sólo para el almacenamiento de variables. Cuando se tienen declaradas variables tipo apuntador, no están restringidas a un área segura como las variables tipo estándar y pueden contener la dirección de cualquier punto de la memoria. **La ejecución de un programa que contenga variables tipo apuntador puede resultar en una catástrofe si el programador no está consciente de lo que está haciendo con tales variables.**

Ahora bien, cuando se tienen que declarar  $n$  variables se tendrá una vecindad donde se localizarán las variables. Observe el programa 2.3:



```
#include <stdio.h>
main()
{
    int x=100;
    double y=4587.215;
    long z=2548261;
    printf("%u      %u      %u\n",&x,&y,&z);
    printf("%d      %d      %d\n", sizeof(x),sizeof(y),sizeof(z));
    getchar();
}
```

```
65524      65516      65512
2          8          4
```

Programa 2.3

La función sizeof() retorna como resultado el número de bytes que ocupa cada una de las variables que se envían como parámetro en la función.

En la ayuda de Turbo C se tienen el siguiente número de bytes por variable:

unsigned char	1 byte
char	1 byte
unsigned int	2 bytes
short int	2 bytes
int	2 bytes
unsigned long	4 bytes
long	4 bytes
float	4 bytes
double	8 bytes
long double	10 bytes

En la primera hilera se muestra la dirección de memoria en donde se encuentra cada una de las tres variables, en la segunda hilera indica el número de bytes de cada tipo de variable. Observe que el apuntador está apuntando al primer byte de cada variable y que las variables se localizan en la memoria en forma continua (ver figura 2.1).

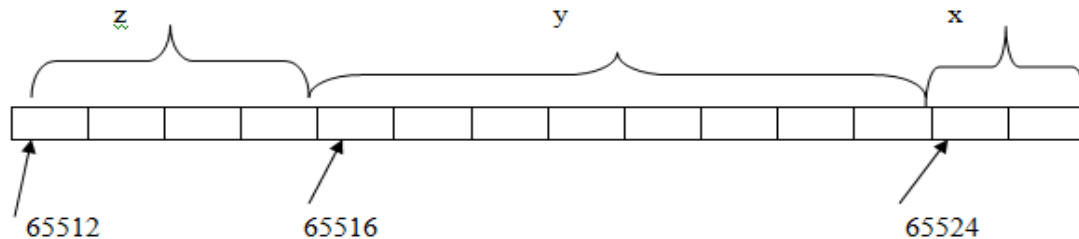
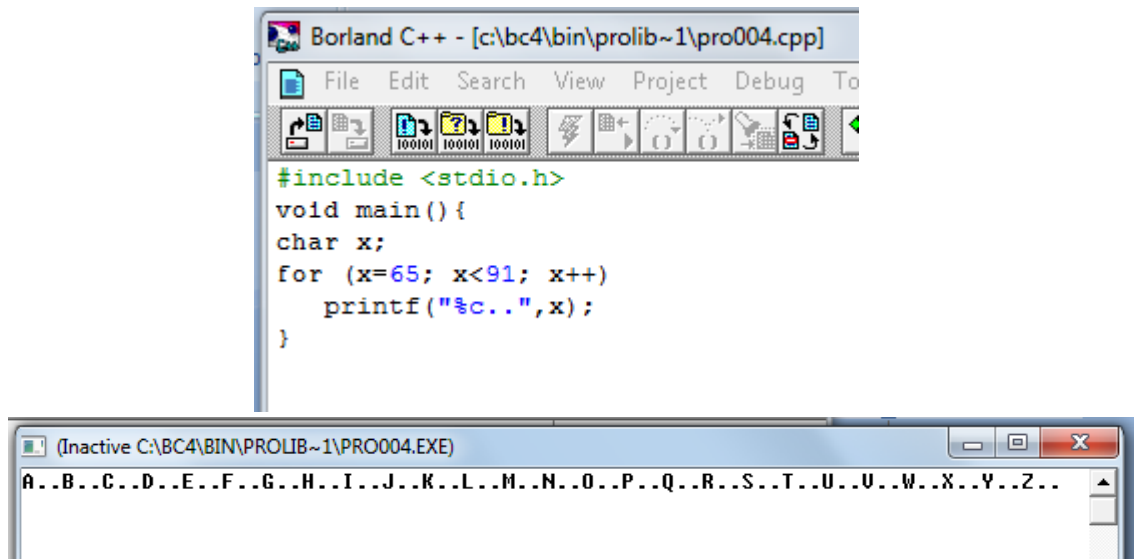


Figura 2.1

## 2.2 Arreglos de caracteres.

Cuando se trata con cadenas de caracteres en C, nos acercamos a operaciones puras de apuntadores. Sin embargo, cadenas en C, o, arreglos de caracteres, también tienen una similitud con números y variables del tipo numérico. La creación y uso de arreglos de caracteres es la siguiente discusión en apuntadores.

El programa 2.4 despliega en el monitor todas las letras del abecedario:



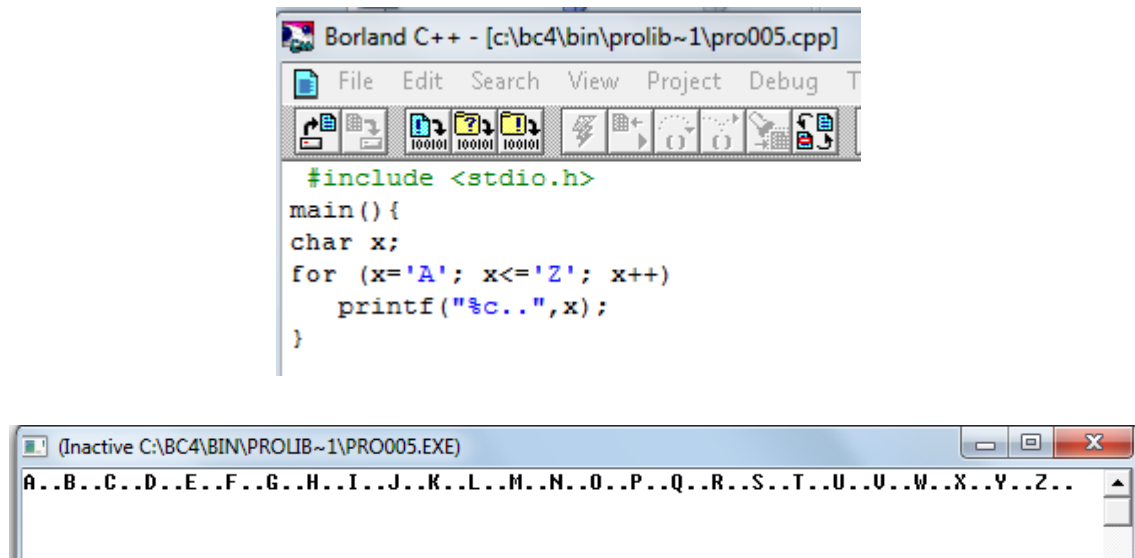
```
#include <stdio.h>
void main() {
    char x;
    for (x=65; x<91; x++)
        printf("%c..",x);
}
```

(Inactive C:\BC4\BIN\PROLIB~1\PRO004.EXE)

A..B..C..D..E..F..G..H..I..J..K..L..M..N..O..P..Q..R..S..T..U..V..W..X..Y..Z..

Programa 2.4

La línea de declaración muestra a una variable – x - del tipo carácter. Esto significa que un simple byte es reservado para el uso exclusivo de esta variable. El valor estándar para una variable del tipo carácter consiste de un entero corto sin signo del rango 0 a 255. El valor 65 en decimal representa la – A - en ASCII. También se puede ejecutar el programa como lo muestra el programa 2.5:



```
#include <stdio.h>
main() {
    char x;
    for (x='A'; x<='Z'; x++)
        printf("%c..",x);
}
```

(Inactive C:\BC4\BIN\PROLIB~1\PRO005.EXE)

A..B..C..D..E..F..G..H..I..J..K..L..M..N..O..P..Q..R..S..T..U..V..W..X..Y..Z..

Programa 2.5

En el lenguaje C, x= 'A' significa exactamente lo mismo que x=65. 'A' se traduce en compilador como 65.

La única razón por la cual se despliega la letra 'A' en la pantalla en vez de un número se debe a la designación de %c en la función printf(). Esto especifica que el valor en su argumento – x -, se desplegará como un carácter ASCII.

De la misma forma analizada con los enteros, la siguiente línea retorna la dirección de almacenamiento de la variable – x -:

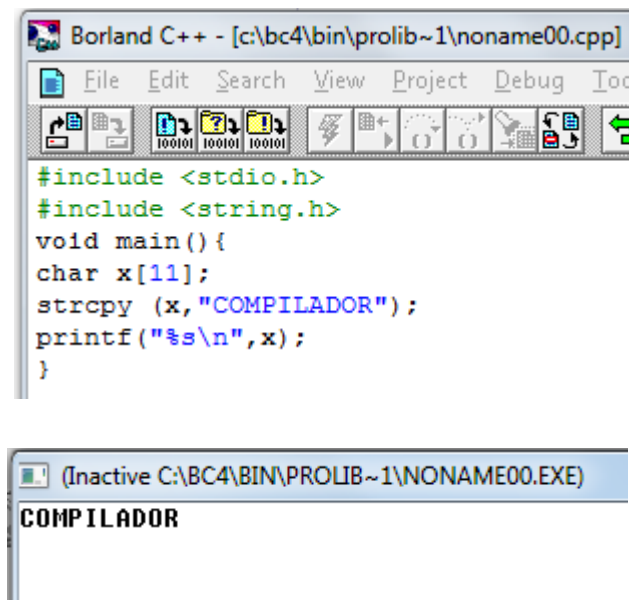
```
printf("%u\n", &x);
```

Si se incluye la siguiente línea, lo que aparecerá en pantalla será el tamaño de almacenamiento de la variable – x -, siendo ésta un carácter, da como resultado el valor de uno:

```
printf("%d\n", sizeof(x));
```

Es extraño observar dentro de un programa la declaración de una variable tipo carácter, lo más común es utilizar un arreglo de caracteres (o cadena de caracteres). Una cadena de caracteres se maneja como una simple unidad en la mayoría de los lenguajes de cómputo, sin embargo, C no tiene una variable tipo cadena de caracteres. En C todas las cadenas de caracteres son guardadas como un arreglo de caracteres donde cada carácter se almacena en un byte. En el programa 2.6, al ejecutarse el programa se desplegará en la pantalla la palabra –COMPILADOR-,

:



Programa 2.6

Con la declaración de la variable – x -, se coloca un espacio en memoria para once bytes. La razón por la que se solicitan 11 bytes cuando la palabra sólo tiene 10 bytes es la siguiente: En C se define como cadena de caracteres a una unidad de caracteres terminados en el

carácter NULL. El carácter NULL en ASCII es el valor cero (ocho bits con el valor de cero). La función `strcpy()` también copia el carácter NULL por lo que se guarda en memoria es la palabra `-COMPILADOR\0-` donde `'\0'` es el carácter nulo y la marca de fin de cadena de caracteres.

En este caso, la función **`strcpy()`** (string copy) copiará la constante de la cadena que se encuentra como parámetro de la derecha a la variable colocada como parámetro a la izquierda hasta encontrar el carácter nulo.

La función `printf("%s\n", x)` escribe en pantalla la cadena de caracteres hasta encontrar el valor nulo. Si por alguna razón la cadena de caracteres no tiene como límite de cadena al valor nulo, se imprimirá basura hasta encontrar un valor nulo. Otro detalle importante, ¿Cómo sabe el compilador desde donde escribir? En este caso, el nombre de la variable `-x-` es un apuntador a la dirección de memoria del primer carácter de la cadena. El operador unario ampersand no puede ser usado en un constructor de arreglo:

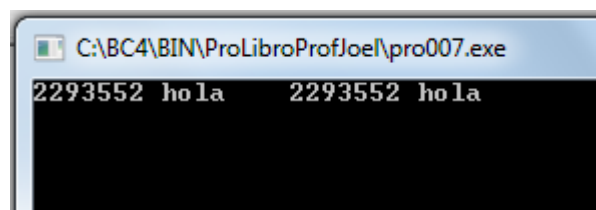
`&x`

Porque `-x-` en sí misma es ya un apuntador. Sin embargo, si se puede utilizar si se utiliza con subíndices.

`&x[0]`

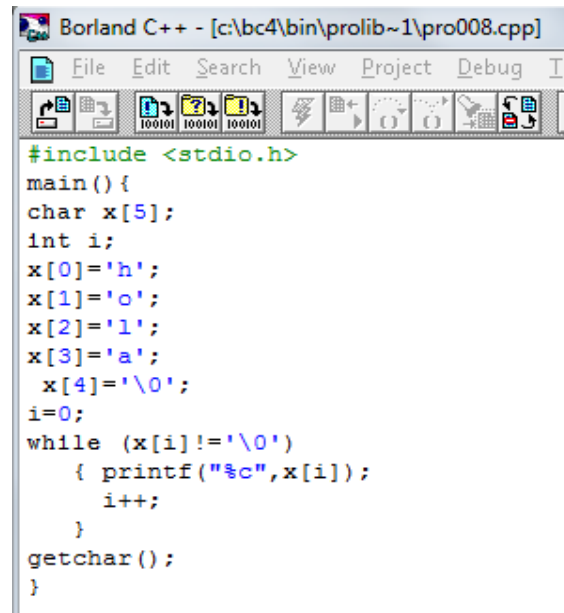
En consecuencia `x[0]` es una variable del tipo carácter y `&x[0]` retorna la dirección de memoria de esta variable, la dirección del primer byte del arreglo. Observe el programa 2.7:

```
#include <stdio.h>
#include <string.h>
main() {
char x[5];
strcpy(x, "hola");
printf("%u %s %u %s\n", &x, x, &x[0], &x[0]);
getchar();
}
```



Programa 2.7

Por lo tanto – **x** - y - **&x[0]** - son uno y el mismo. El valor que se muestre en su computadora puede variar, sin embargo se obtendrán dos valores iguales indicando el inicio del arreglo. De esta forma **&x[1]** mostrará la dirección del segundo byte del arreglo, etc. Otra forma de guardar información en un arreglo se muestra en el programa 2.8:



```

#include <stdio.h>
main() {
char x[5];
int i;
x[0]='h';
x[1]='o';
x[2]='l';
x[3]='a';
x[4]='\0';
i=0;
while (x[i]!='\0')
{ printf("%c",x[i]);
i++;
}
getchar();
}

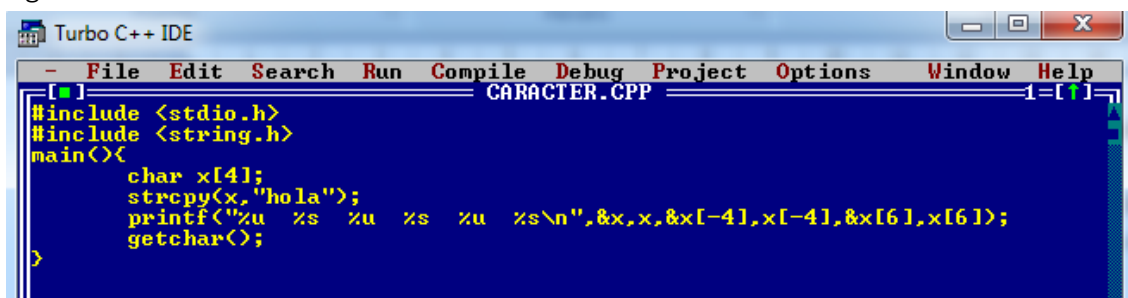
```

Programa 2.8

El ciclo while muestra en forma muy general como realiza su trabajo la función printf() para %s.

Recuerde que la variable – **x** - no es igual a “hola”. Sólo retorna la dirección de memoria donde se localiza el primer carácter del arreglo.

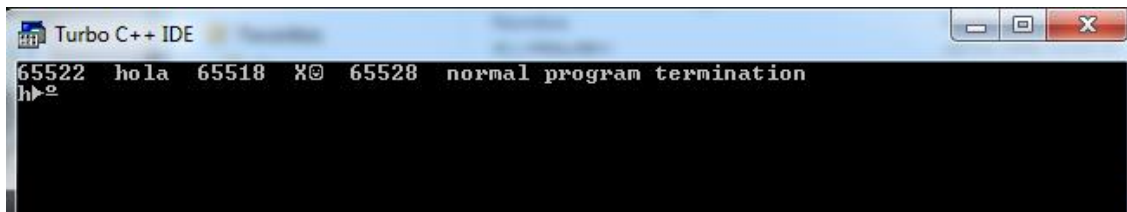
Una debilidad de C es que no tiene una forma de verificar los límites del arreglo, por lo que se puede desbordar a la izquierda como en la derecha y el compilador no verifica el desbordamiento (otros lenguajes como: lo son Pascal, Java, PL/1, etc, verifican en el momento de la compilación y además en la ejecución. Si existe un desbordamiento en un arreglo, envía un mensaje de error). Un ejemplo de desbordamiento se muestra en el programa 2.9:



```

#include <stdio.h>
#include <string.h>
main() {
char x[4];
strcpy(x,"hola");
printf("%u %s %u %s %u %s\n",&x,x,&x[-4],x[-4],&x[6],x[6]);
getchar();
}

```



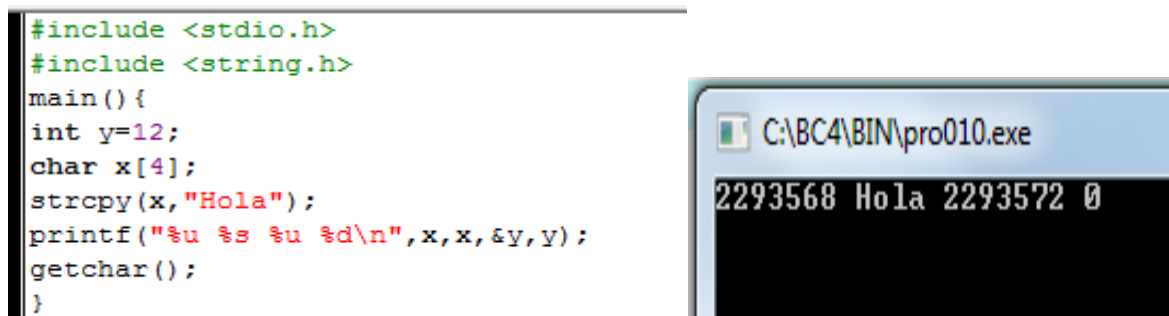
```
Turbo C++ IDE
65522 hola 65518 %0 65528 normal program termination
h>
```

Programa 2.9

Donde:

- `-%x-` y `-%x-` muestran la dirección y contenido del arreglo.
- `-%x[-4]-` muestra la dirección cuatro bytes antes de donde se encuentra la palabra "hola" y posteriormente muestra el contenido que es sólo basura.
- `-%x[6]-` muestra una dirección 6 bytes después del inicio del arreglo declarado y muestra lo que se tenga en esa posición que puede ser basura, en este caso guardó el comentario "normal programa termination" con salto de línea y basura hasta encontrar el fin de línea `"\0"`.
- Al ejecutar el programa en una ocasión diferente, se pueden mostrar diferentes elementos.
- Si se ejecuta con otro compilador, puede en un momento dado producir error.

La falta de comprobación de límites simplemente significa que no existe una forma de protección para que no exista una sobre escritura en cualquier arreglo. Por ejemplo, en el programa 2.10 muestra en pantalla:



```
#include <stdio.h>
#include <string.h>
main() {
    int y=12;
    char x[4];
    strcpy(x, "Hola");
    printf("%u %s %u %d\n", x, x, &y, y);
    getchar();
}
```

```
C:\BC4\BIN\pro010.exe
2293568 Hola 2293572 0
```

Programa 2.10

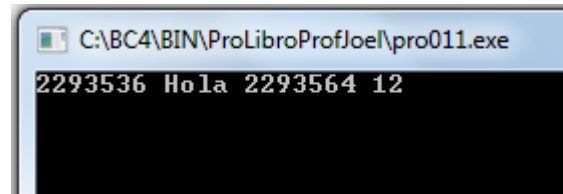
El programa 2.11 muestra en pantalla:



```

#include <stdio.h>
#include <string.h>
main() {
    int y=12;
    char x[5];
    strcpy(x, "Hola");
    printf("%u %s %u %d\n", x, x, &y, y);
    getchar();
}

```



Programa 2.11

En ambos programas se trata de guardar en la variable – x - la palabra “Hola”. Para poder guardar la palabra “Hola” se requieren 5 bytes porque se tiene que guardar también el carácter nulo (“Hola\0”). En consecuencia, en el primer programa se sobre escribió el valor del arreglo ocupando uno de los dos bytes pertenecientes a la variable entera – y -, de esta forma la variable entera muestra en pantalla un cero en lugar de un doce. (Pruebe con los siguientes valores y= 255, y= 256).

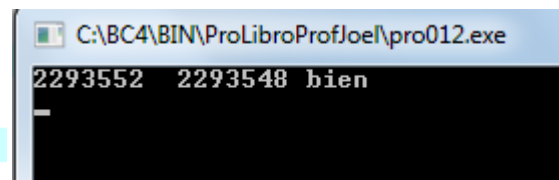
Cuando el número de caracteres rebasa al tamaño del arreglo declarado existe la posibilidad de que el arreglo invada memoria que ha sido dada a otras variables por lo que puede sobrescribir causando una secuencia impropia en la ejecución. Un caso extremo puede incluir a porciones de la administración de memoria produciendo un conflicto siendo necesaria la inicialización de la ejecución del programa o hasta del mismo sistema operativo. O puede realizar borrado de disco duro por una inadvertida escritura a direcciones de interrupción, produciendo un desastre.

Otro ejemplo se muestra en el programa 2.12 con lo que aparece en pantalla:

```

#include <stdio.h>
#include <string.h>
main() {
    char x[10];
    char y[4];
    strcpy(x, "como estas");
    strcpy(y, "bien");
    printf("%u %s %u %s\n", &x, x, &y, y);
    getchar();
}

```



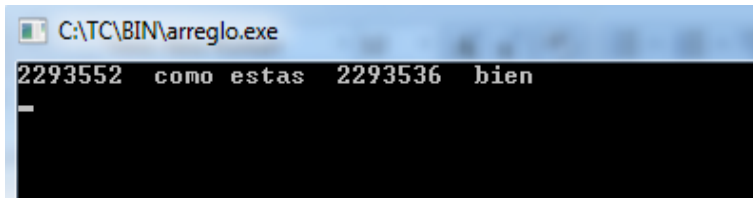
Programa 2.12

Si hacemos un pequeño cambio se obtiene el programa 2.13, observe la impresión en pantalla:

```

#include <stdio.h>
#include <string.h>
main() {
    char x[10];
    char y[5];
    strcpy(x, "como estas");
    strcpy(y, "bien");
    printf("%u %s %u %s\n", &x, x, &y, y);
    getchar();
}

```



Programa 2.13

En este programa, la sobre posición no es desastrosa. Adicionalmente, el error fue fácilmente reconocido y fácilmente resuelto incluyendo más espacio en una de las variables. Sin embargo, si se tiene un programa mucho más complejo, una sobrescritura puede producir horas de revisión o debugging.

Ejercicios:

- Explique la diferencia de las dos sentencias indicadas a continuación:
  - `printf("%d", x);`
  - `printf("%u", &x);`
- Explique la función `sizeof()` y de un ejemplo.
- Explique lo que imprime el siguiente código:
 

```
for (int i=48; i<58; i++)
    printf("%c..", i);
```
- Explique la forma en que se guarda en memoria un tipo de dato abstracto `int` en lenguaje de programación C.
- ¿Para qué sirve la función `strcpy()`?
- En el código: `- char x[10]; -`.  
Indicar que tipo de variable es `-x-`.
- En el código: `- char x[10]; -`.  
Indicar la diferencia o semejanza entre `-x-` y `-x[0]-`.
- En el código: `- char x[10]; -`.  
Comente si es posible o no realizar la siguiente instrucción:  
`putchar(x[-4]);`

## TERCER CAPITULO: \*, Una variable tipo apuntador.

### Resumen.

Este capítulo forma parte de la primera unidad de competencia en la parte de “Variables dinámicas: Apuntadores, Operaciones básicas” y muestra el segundo operador básico “\*”, que permite modificar valores en la dirección de memoria donde se localiza una variable. Es fundamental comprender el uso de tal operador ya que nos permite la modificación de valores en memoria por medio de un “alias” tanto en variables creadas en forma estática como variables creadas en forma dinámica en lo que es la memoria principal.

Los programas son de propia autoría y las ideas básicas se tomaron de los siguientes libros:

(Deitel & Deitel, 1995)

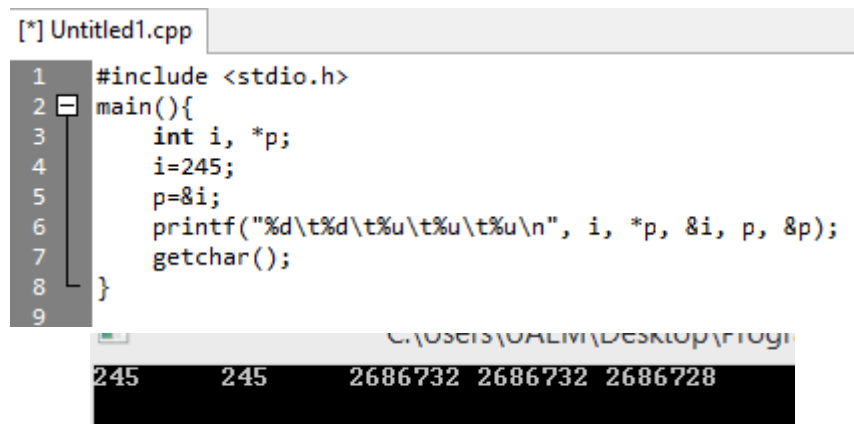
(Ceballos, 2015)

(Backman, 2012)

### 3.1 Variables tipo apuntador y operadores básicos.

Los apuntadores analizados hasta este momento tienen un valor fijo o constante; es decir, están unidos a una variable que ha sido declarada en algún lugar del programa. En este capítulo se describen variables que han sido declaradas como apuntadores. Estas variables no guardan objetos, al menos como se cree en relación con otras variables. Las variables tipo apuntador guardan la dirección de una locación de memoria. A comparación de los apuntadores analizados previamente, una variable puede apuntar a cualquier área de la memoria.

El programa 3.1 permitirá iniciar la explicación de la declaración de variables tipo apuntador en el lenguaje de programación C. Al ejecutar el programa, los resultados varían dependiendo del compilador y de la computadora, lo que se muestra en pantalla es lo siguiente:



```
[*] Untitled1.cpp
1  #include <stdio.h>
2  main(){
3      int i, *p;
4      i=245;
5      p=&i;
6      printf("%d\t%d\t%u\t%u\t%u\n", i, *p, &i, p, &p);
7      getchar();
8  }
9
```

C:\Users\OALM\Desktop\Progi

245      245      2686732 2686732 2686728

Programa 3.1

En la línea de declaración se tienen dos variables, una de ellas es una variable automática del tipo entero (int i) y la otra es una variable apuntador de tipo entero (int \*p). El operador unario asterisco (\*) es la clave en la declaración.

- \*p Es una variable del tipo entero.
- p Apuntará a una localidad de memoria del tipo entero.

Para que \*p muestre un valor de tipo entero, primero se le debe de indicar la localidad de memoria a la que se está haciendo referencia por lo que al realizar la igualdad  $p = \&i$ , la variable  $p$  toma el valor de la dirección de memoria de la variable  $i$ . Ahora bien,  $*p$  mostrará el contenido de la localidad de memoria de la variable  $i$  y  $\&p$  mostrará la dirección de la localidad de memoria de la variable  $p$ . En la figura 3.1 muestra el direccionamiento de memoria de una ejecución con Dev-C(ver figura 3.1).

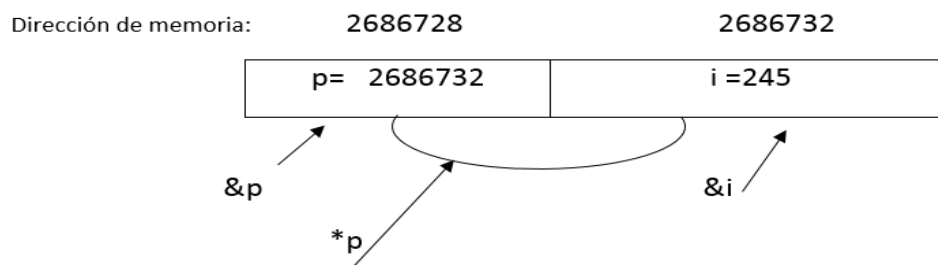


Figura 3.1

Por lo tanto:

$$\&i \equiv p \quad \text{y} \quad *p \equiv i.$$

Se cumple  $*p \equiv i$  *si y solo si* primero se realizó la asignación  $p = \&i$ .

Hasta este momento hemos visto cómo una variable del tipo apuntador puede ser utilizada para retornar un valor de una localidad de memoria a la cual está siendo direccionada. El programa 3.2 se muestra cómo la variable de tipo apuntador se utiliza para modificar el valor de la localidad de memoria a la que hace referencia:

```

c:\bc4\bin\prolib~1\joel2002.cpp
#include <stdio.h>
main() {
    int i, *p;
    i=345;
    p=&i;
    *p=485;
    printf("%d\n", i);
}

```

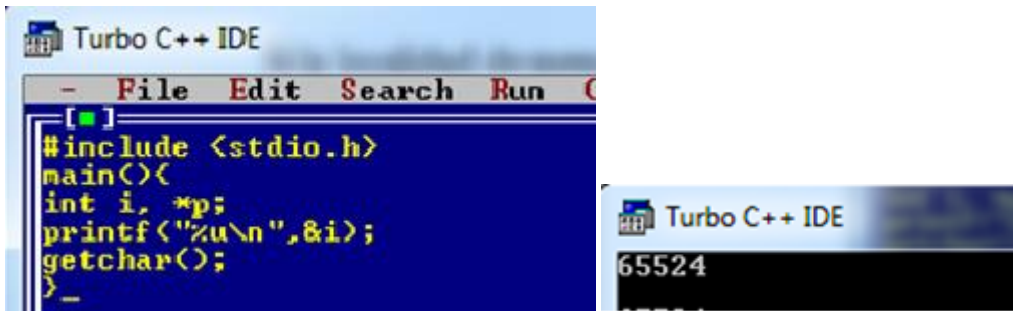
(Inactive C:\BC4\BIN\PROLIB~1\JOEL2002.EXE)

485

Programa 3.2

Observe que el valor asignado a la variable – i - es 345, posteriormente se le asigna al apuntador –p- la dirección de la localidad de memoria de la variable – i -. En la siguiente línea, la conexión del operador unario \*- al apuntador –p- da como resultado al acceso de la localidad donde hace referencia –p-, por lo que el valor 345 es sobre escrito colocándose el valor 485.

Si la localidad de memoria se conoce, la asignación se puede realizar en forma directa. Ejecutemos el programa 3.3:



```

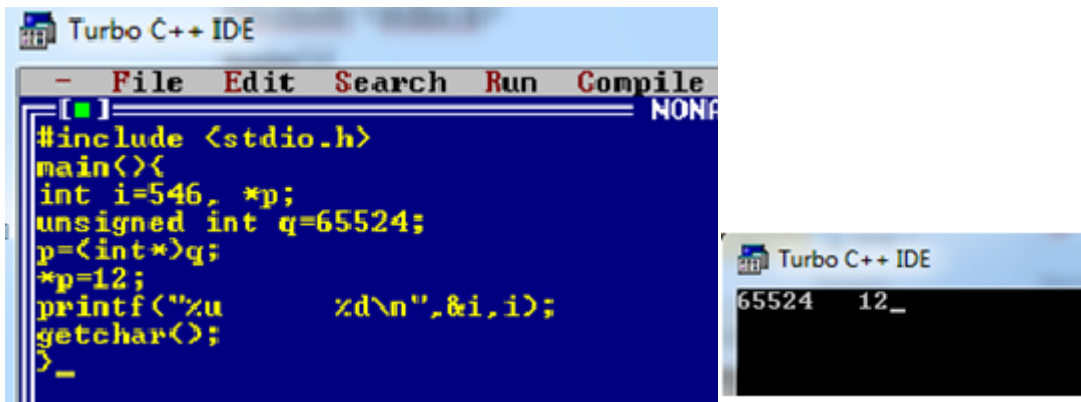
Turbo C++ IDE
- File Edit Search Run C
[ ]
#include <stdio.h>
main(){
int i, *p;
printf("%u\\n",&i);
getchar();
} _
65524

```

Programa 3.3

Observe que la variable – i - se encuentra en la localidad 65524.

Ahora, en la misma computadora, inmediatamente ejecute el programa 3.4:



```

Turbo C++ IDE
- File Edit Search Run Compile
[ ]
#include <stdio.h>
main(){
int i=546, *p;
unsigned int q=65524;
p=<int*>q;
*p=12;
printf("%u %d\\n",&i,i);
getchar();
} _
65524 12_

```

Programa 3.4

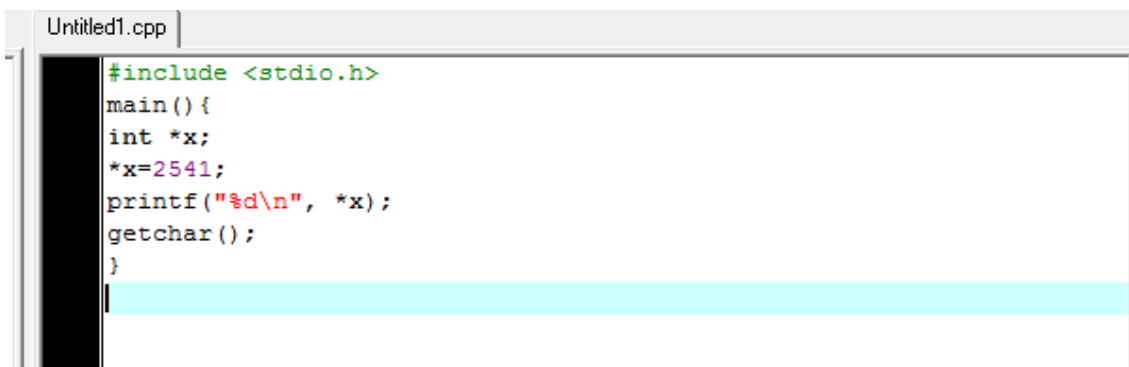
Esto es porque al apuntador se le asigna la dirección de memoria de la localidad de la variable – i -. Al momento de la asignación se debe de realizar un casting (o moldeado) de una constante entera sin signo a un apuntador tipo entero ya que de otra forma se produce un error de tipo mismatch (los tipos de variable no coinciden). (Este programa en particular fue ejecutado con TC.)

Por lo que:

```
unsigned q=65524;
```

`p=(int *)q ≡ p=&i //Para una ejecución específica`  
(El utilizar una variable del tipo unsigned es para poder tener el rango de 0-65536)

Como se recordará, una variable tipo apuntador no se inicializa, por lo que en su localidad de memoria puede apuntar a cualquier dirección de la memoria (basura), uno debe tener la certeza de que el área de memoria es segura. Es decir, que no debe ser utilizada para otros propósitos del cual el programador no pueda tener conocimiento. El programa 3.5 muestra un peligroso uso de un apuntador:



```
#include <stdio.h>
main() {
    int *x;
    *x=2541;
    printf("%d\\n", *x);
    getchar();
}
```

Programa 3.5

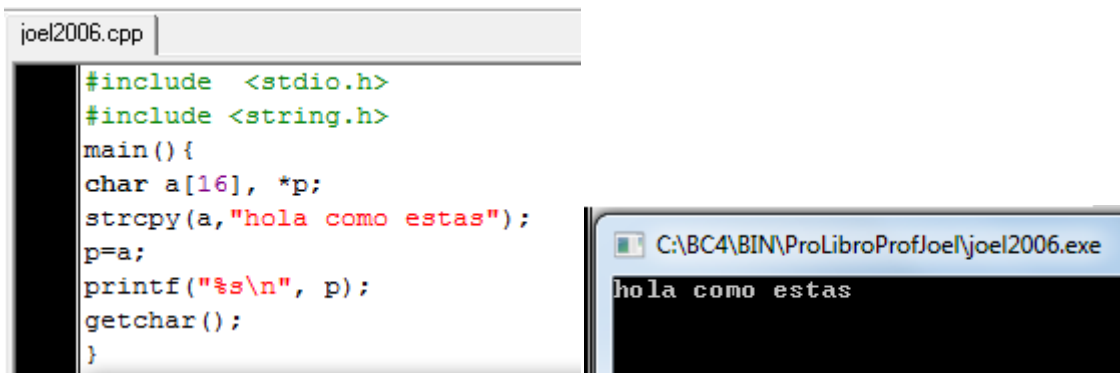
En este caso, se declara un apuntador del tipo entero, pero no se inicializa con una dirección segura específica. Por lo que puede contener cualquier dirección de memoria. El cuestionamiento es: ¿a qué o adonde apunta `*x`?

Cuando se declara una variable y no se inicializa, ésta contiene un valor aleatorio. Esto mismo sucede con los apuntadores, si el apuntador no se inicializa puede apuntar a cualquier área de memoria. Si el programa es tan pequeño como el del ejemplo, puede ser que no suceda nada fuera de lo usual y el programa se ejecute como lo esperado. Existe la posibilidad de que la computadora sea reiniciada o que falle sin alguna razón.

### 3.2 Variables tipo apuntador para cadenas de caracteres.

Los apuntadores tipo carácter se utilizan con frecuencia para tener acceso a cadenas de caracteres. **Cadenas de caracteres y apuntadores a carácter pueden ser tratados en forma indistinta.** En el capítulo anterior se comentó que el nombre de un arreglo de caracteres (sin el uso de un subíndice) se considera como apuntador al primer elemento del arreglo de caracteres.

El programa 3.6 se muestra la declaración de un apuntador tipo carácter y su uso potencial:



```
joel2006.cpp
#include <stdio.h>
#include <string.h>
main() {
    char a[16], *p;
    strcpy(a, "hola como estas");
    p=a;
    printf("%s\n", p);
    getchar();
}
```

C:\BC4\BIN\ProLibroProfJoel\joel2006.exe

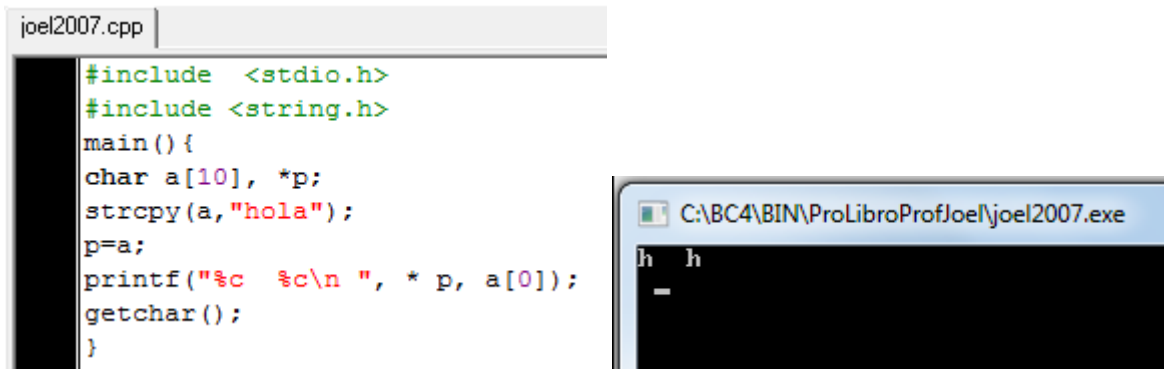
hola como estas

Programa 3.6

Al ejecutar el programa se muestra que la variable `-p-` apunta al inicio de la secuencia de diez caracteres. Observe que el operador ampersand no se utilizó para el acceso a la dirección de memoria del primer elemento ya que la variable `-a-`, cuando se utiliza sin subíndice, es a la vez el nombre del arreglo y el apuntador al primer elemento del arreglo. En consecuencia:

$$p=a \quad \equiv \quad p=\&a[0]$$

La variable `-p-` apunta al primer elemento de la variable, entonces `-*p-` muestra lo que contiene en el primer elemento del arreglo. Observe el programa 3.7:



```
joel2007.cpp
#include <stdio.h>
#include <string.h>
main() {
    char a[10], *p;
    strcpy(a, "hola");
    p=a;
    printf("%c %c\n ", *p, a[0]);
    getchar();
}
```

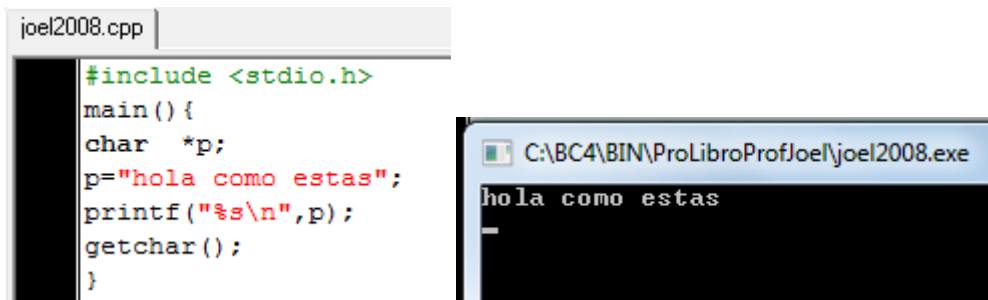
C:\BC4\BIN\ProLibroProfJoel\joel2007.exe

h h  
-

Programa 3.7

Se puede decir que el apuntador apunta a un área segura que es compuesta de 10 bytes consecutivos.

Una forma más sencilla de escribir el primer programa y sin la necesidad de declarar arreglos se muestra en el programa 3.8:

The image shows a code editor window titled 'joel2008.cpp' on the left and a console window titled 'C:\BC4\BIN\ProLibroProfJoel\joel2008.exe' on the right. The code in the editor is:

```
#include <stdio.h>
main() {
    char *p;
    p="hola como estas";
    printf("%s\n",p);
    getchar();
}
```

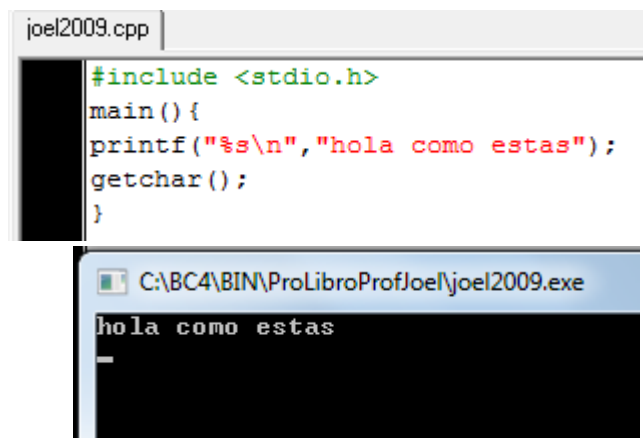
The console window shows the output 'hola como estas' followed by a cursor line.

Programa 3.8

El programa 3.8 realiza lo mismo que el programa 3.6, pero sin la necesidad de la declaración del arreglo de caracteres, además de no necesitar la función strcpy() y la librería string.h.

Pero se asignó la dirección de un objeto a un apuntador y el apuntador no fue inicializado. El punto por analizar es porque el programa es correcto, recordemos que la prioridad de una asignación es de derecha a izquierda, por lo que la primer acción a realiza es identificar el elemento que está a la derecha de la igualdad, siendo ésta una cadena de caracteres constante se le asignará un espacio de memoria que es la dirección del inicio de este espacio al que se le asignará a la variable `p`. Por lo que la variable `p` apunta a un área segura de memoria, el área donde la constante fue almacenada.

El programa descrito es casi lo mismo al programa 3.9:

The image shows a code editor window titled 'joel2009.cpp' on the left and a console window titled 'C:\BC4\BIN\ProLibroProfJoel\joel2009.exe' on the right. The code in the editor is:

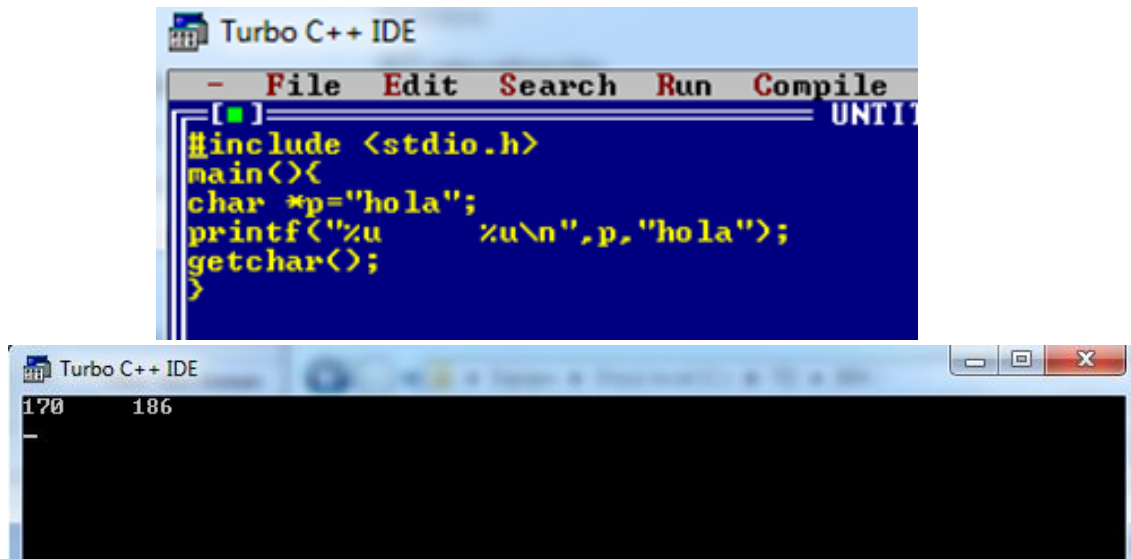
```
#include <stdio.h>
main() {
    printf("%s\n", "hola como estas");
    getchar();
}
```

The console window shows the output 'hola como estas' followed by a cursor line.

Programa 3.9

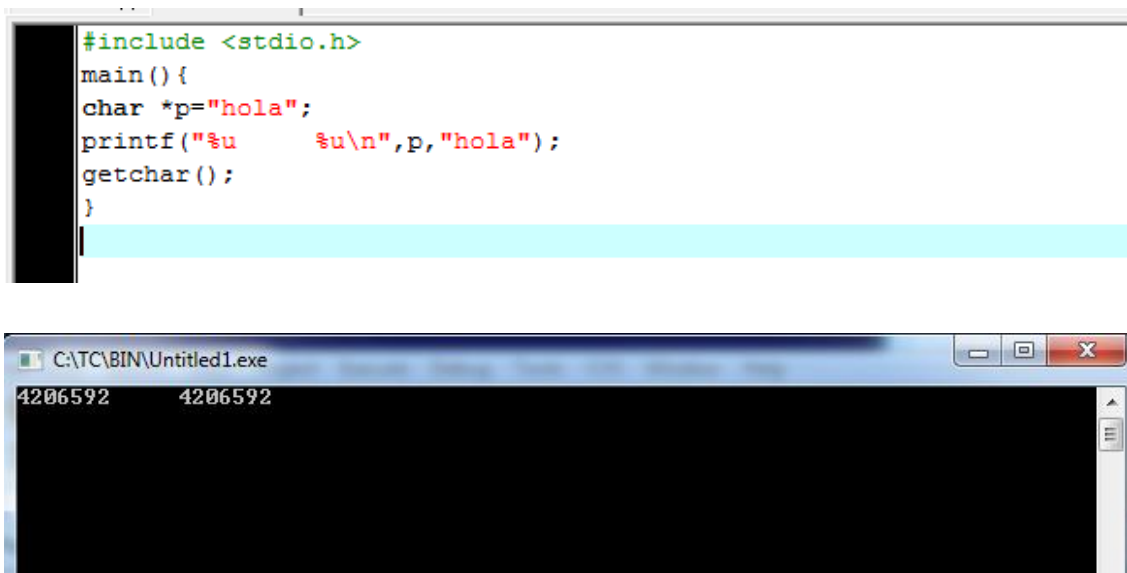
La forma de ver el espacio de las constantes depende del compilador, por ejemplo, se muestra el programa 3.10A en TC:





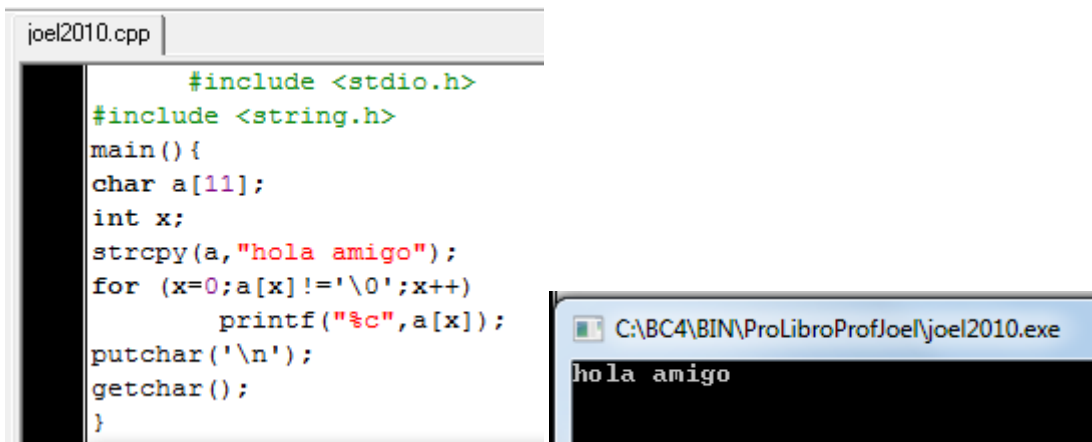
Programa 3.10A

Se podría pensar que al ejecutarse el programa se desplegaría en el monitor la misma dirección de memoria, lo que realmente sucede es que existen dos constantes con el mismo contenido, el compilador destinará dos direcciones diferentes de memoria aunque se tenga el mismo contenido, la variable `p` apuntará al inicio de una constante y al momento de la invocación de la función `printf()` se asignará el espacio de memoria a la segunda constante. Observe la ejecución del mismo programa en DEV-C (programa 3.10B) y opine:



Programa 3.10B

Ahora se observará otra forma del manejo de apuntadores utilizando los ejemplos de los programas 3.11 y 3.12.

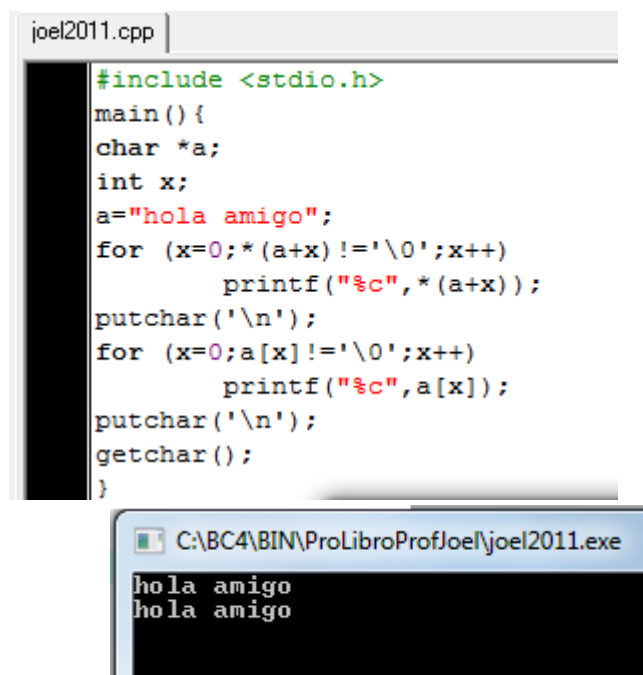


```
joel2010.cpp
#include <stdio.h>
#include <string.h>
main() {
char a[11];
int x;
strcpy(a, "hola amigo");
for (x=0; a[x]!='\0'; x++)
    printf("%c", a[x]);
putchar('\n');
getchar();
}
```

C:\BC4\BIN\ProLibroProfJoel\joel2010.exe  
hola amigo

Programa 3.11

En este ejemplo se observa cómo se puede imprimir cada uno de los caracteres utilizando un operador de repetición como es el “for”.



```
joel2011.cpp
#include <stdio.h>
main() {
char *a;
int x;
a="hola amigo";
for (x=0; *(a+x)!='\0'; x++)
    printf("%c", *(a+x));
putchar('\n');
for (x=0; a[x]!='\0'; x++)
    printf("%c", a[x]);
putchar('\n');
getchar();
}
```

C:\BC4\BIN\ProLibroProfJoel\joel2011.exe  
hola amigo  
hola amigo

Programa 3.12

Sabiendo que la variable del tipo char sólo ocupa un byte, se puede manejar el comando “for” con un incremento de uno en uno. Ahora bien, se sabe que la variable **a** apunta al primer elemento de la cadena de caracteres, pero ¿si se le suma uno al apuntador? entonces se estaría apuntando al siguiente elemento del arreglo, por lo tanto:

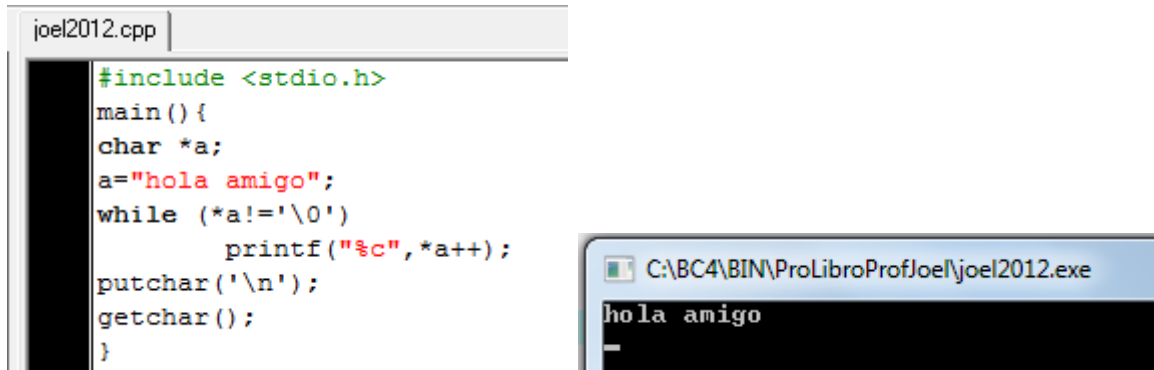
- \*a ≡ a[0]
- \*(a+1) ≡ a[1]
- \*(a+2) ≡ a[2]

$*(a+3) \equiv a[3]$

Es más eficiente la declaración de apuntadores a carácter que la declaración de arreglo de caracteres. En el arreglo de caracteres se tiene que definir el tamaño de memoria a utilizar, por ejemplo `char b[20]`, esto indica que se tiene que asignar el espacio de 20 bytes aun si no los usa todos o que exista un desbordamiento produciendo los problemas previamente analizados.

Si se usan apuntadores de carácter, no se tienen que reservar una determinada cantidad fija de elementos, esto tiene una gran ventaja, si el apuntador va a hacer referencia a una constante del tipo cadena de caracteres, la cadena puede ser de cualquier tamaño y no se desperdicia espacio o no se corre el riesgo de desbordamiento.

Otra forma para mostrar una cadena de caracteres se observa en el programa 3.13:



The image shows a code editor window titled 'joel2012.cpp' containing the following C++ code:

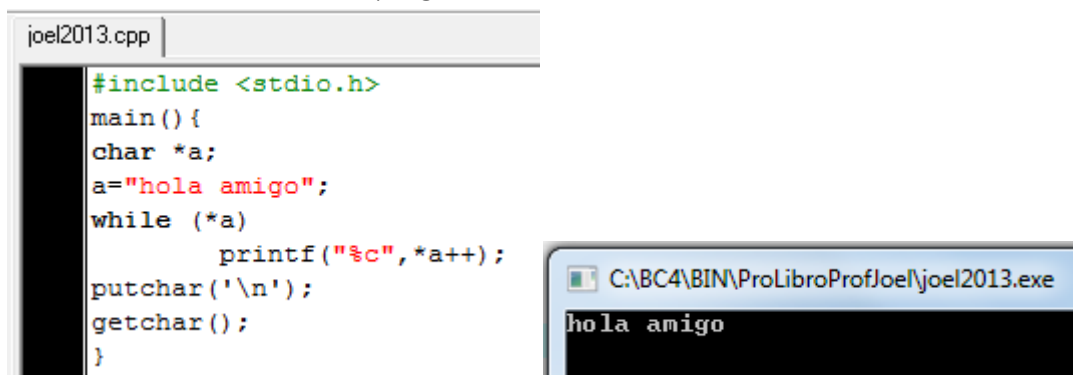
```
#include <stdio.h>
main() {
    char *a;
    a="hola amigo";
    while (*a!='\0')
        printf("%c", *a++);
    putchar('\n');
    getchar();
}
```

To the right of the code editor is a screenshot of a command prompt window titled 'C:\BC4\BIN\ProLibroProfJoel\joel2012.exe'. The output displayed in the command prompt is 'hola amigo' followed by a newline character.

Programa 3.13

Al ejecutarse la función `printf()` despliega en el monitor el contenido de la localidad donde está apuntando la variable `a`, al incrementarse el apuntador en uno, se permite desplegar en el monitor el siguiente carácter hasta que se llega al carácter NULL.

Otra forma se observa en el programa 3.14:



The image shows a code editor window titled 'joel2013.cpp' containing the following C++ code:

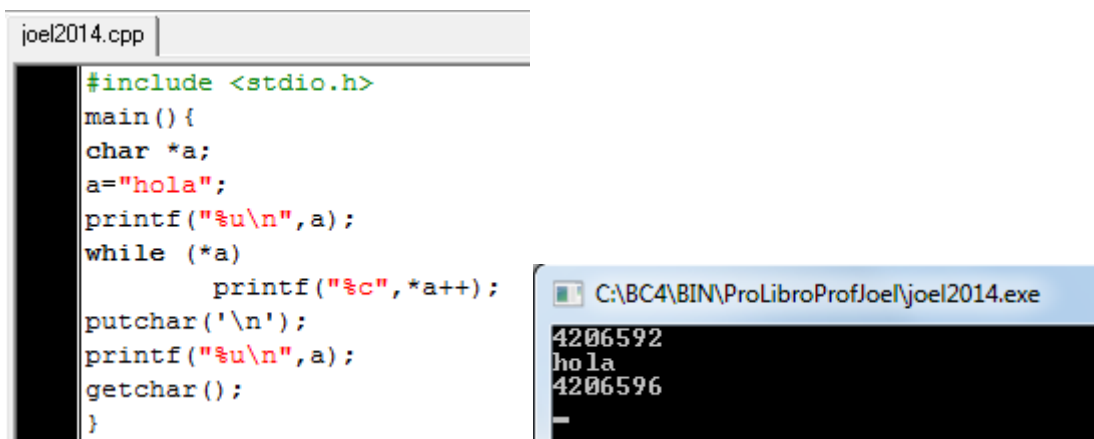
```
#include <stdio.h>
main() {
    char *a;
    a="hola amigo";
    while (*a)
        printf("%c", *a++);
    putchar('\n');
    getchar();
}
```

To the right of the code editor is a screenshot of a command prompt window titled 'C:\BC4\BIN\ProLibroProfJoel\joel2013.exe'. The output displayed in the command prompt is 'hola amigo' followed by a newline character.

Programa 3.14

El operador de control while se ejecuta mientras la condición sea verdadera (es decir, mientras que el resultado de la condición sea diferente a cero) el operador romperá el ciclo cuando la condición sea falsa (o sea, hasta que la condición tenga el valor de cero), al recorrer la cadena al llegar al valor NULL, se tiene el valor de cero como condición produciendo que se rompa el ciclo.

En los dos últimos ejemplos se ha modificado la dirección de memoria del apuntador utilizando el operador unario “++”, por lo que el apuntador, al final, se queda con la dirección de localidad de memoria donde se localiza el valor NULL. Esto se puede probar con el programa 3.15:



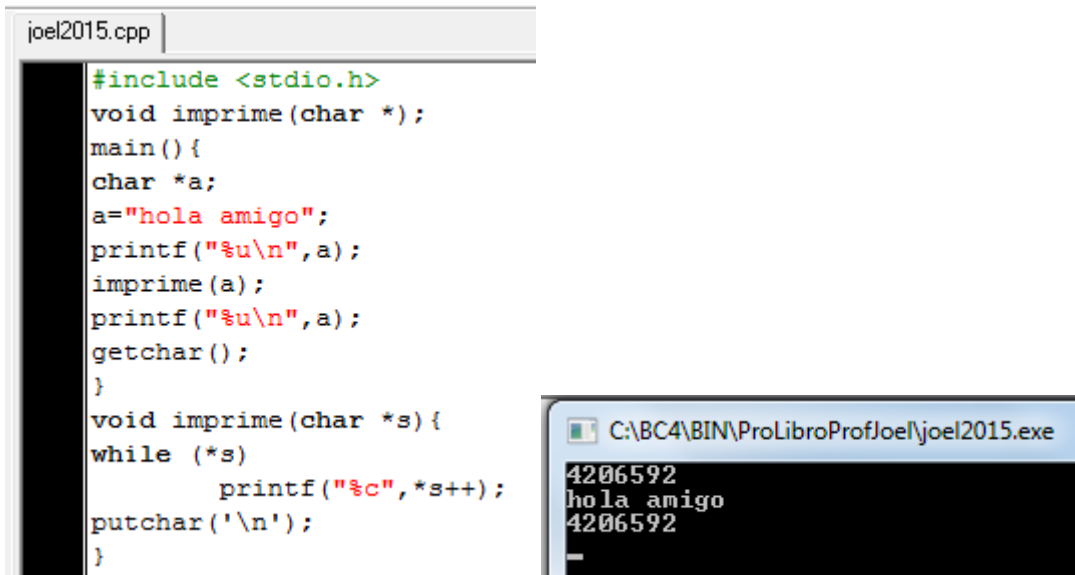
The screenshot shows a code editor with a file named 'joel2014.cpp' and a terminal window showing the execution of 'joel2014.exe'. The code in the editor is as follows:

```
#include <stdio.h>
main() {
    char *a;
    a="hola";
    printf("%u\n",a);
    while (*a)
        printf("%c",*a++);
    putchar('\n');
    printf("%u\n",a);
    getchar();
}
```

The terminal output shows the memory address 4206592, the string 'hola', the memory address 4206596, and a blank line after pressing Enter.

Programa 3.15

Para que el apuntador mantenga la dirección de memoria del inicio de la cadena de caracteres se puede utilizar una función como se muestra en el programa 3.16:



The screenshot shows a code editor with a file named 'joel2015.cpp' and a terminal window showing the execution of 'joel2015.exe'. The code in the editor is as follows:

```
#include <stdio.h>
void imprime(char *);
main() {
    char *a;
    a="hola amigo";
    printf("%u\n",a);
    imprime(a);
    printf("%u\n",a);
    getchar();
}
void imprime(char *s) {
    while (*s)
        printf("%c",*s++);
    putchar('\n');
}
```

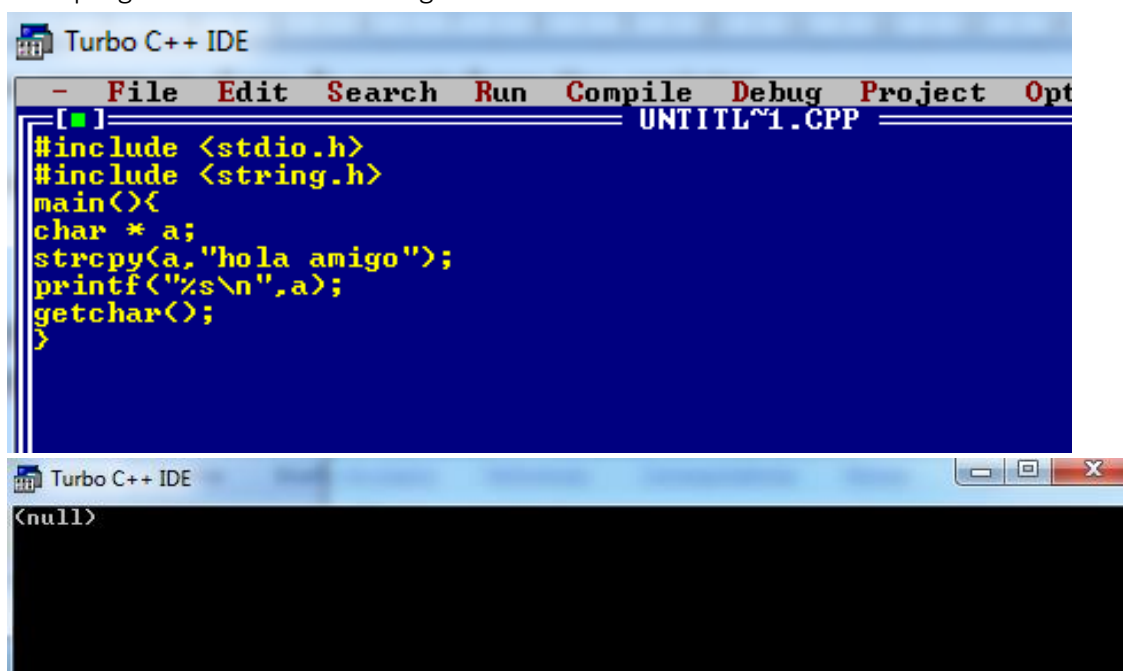
The terminal output shows the memory address 4206592, the string 'hola amigo', the memory address 4206592, and a blank line after pressing Enter.

Programa 3.16

El lenguaje de programación C sólo tiene pase de parámetros por valor, al invocarse la función -imprime()- se envía como parámetro el valor de la dirección del inicio de cadena. Al ejecutarse la función -imprime()- la variable local -s- recibe como valor la dirección de inicio de la cadena. Al finalizar el operador de repetición while(), la variable -s- apuntará a la localidad cuyo valor es NULL y la función imprime() retornará el control a la función donde fue invocada y el valor de la variable -a- no ha sido alterado. Por lo tanto, al ejecutarse el programa lo que se observa en la variable -a- es la misma dirección de memoria antes de invocar la función -imprime()- y después de la invocación a tal función.

### 3.3 Algunos errores en el uso de apuntadores tipo carácter.

En el programa 3.17 se tiene un grave error:



The image shows two windows from the Turbo C++ IDE. The top window, titled 'UNTITL~1.CPP', contains the following C code:

```
#include <stdio.h>
#include <string.h>
main(){
char * a;
strcpy(a,"hola amigo");
printf("%s\n",a);
getchar();
}
```

The bottom window shows the output of the program, which is '<null>'. This indicates a runtime error where the pointer 'a' points to an invalid memory location.

Programa 3.17

En Turbo C, el programa se podrá compilar y ejecutar e imprimirá <null>, en DEV-C se podrá compilar, pero no se podrá ejecutar.

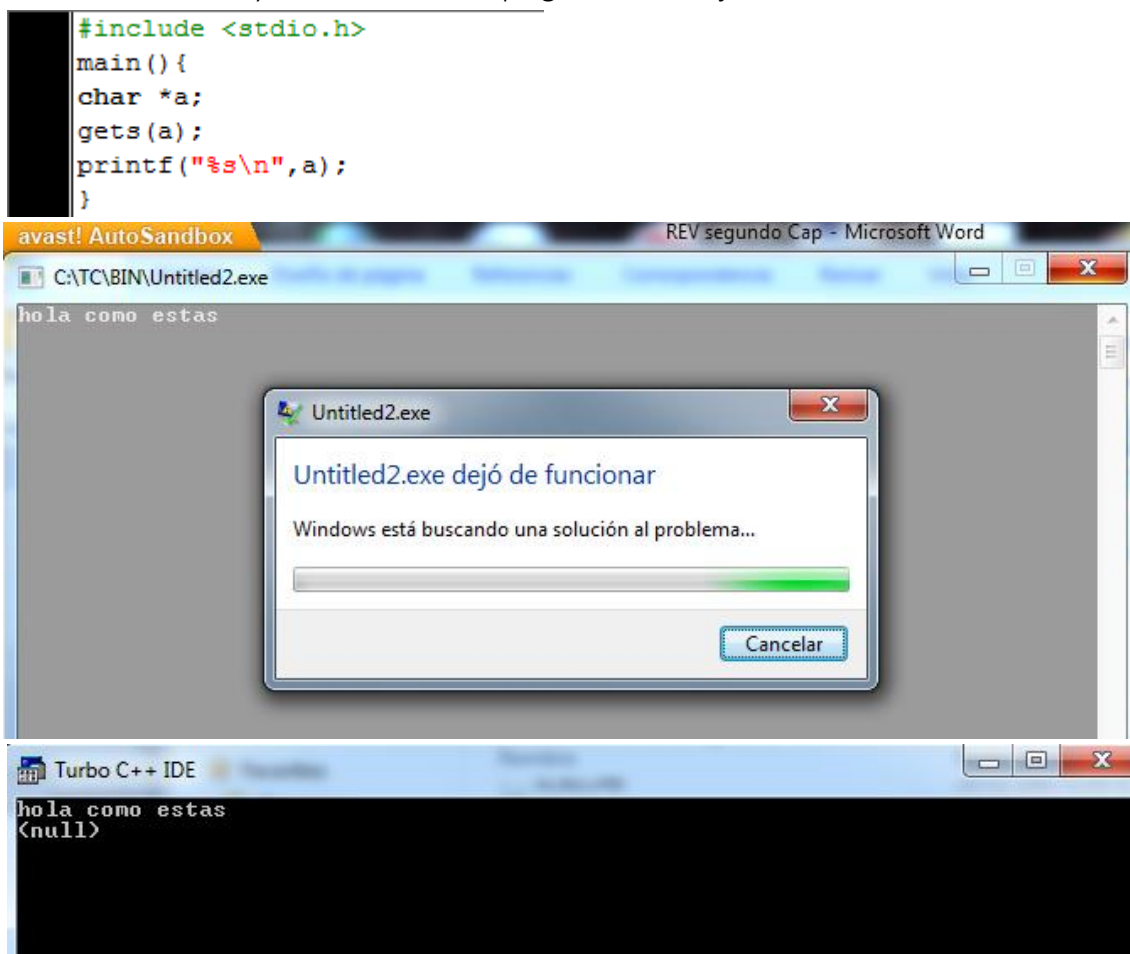
En este ejemplo, cuando se declara la variable -a- se inicializa con un valor aleatorio, al invocar la función strcpy() se asignará la constante "hola amigo" a un lugar de memoria que puede ser inseguro produciendo los conflictos ya mencionados cuando un apuntador no se ha inicializado correctamente. El compilador no encontrará error lexicográfico ni sintáctico, pero en la ejecución se pueden generar tres posibles errores; la primera es que el compilador muestre un valor <null>, la segunda es que el compilador aborte el programa y

muestre un letrero indicando que existió una violación al tratar de escribir en la dirección 0x0. Y la tercer opción es que ejecute sin problema, en esta opción si se tiene un buen antivirus detectará que se trata de sobre escribir en zona insegura y bloqueará el proceso.

En el programa 3.18 se muestra otro posible error. En este programa la función -gets()- leerá caracteres del teclado hasta que se oprima la tecla “enter” o “return”. En ese momento, lo introducido por el teclado será asignado a la variable que fue enviada como parámetro de la función. Se podrá observar que la variable -a- no está inicializada y tiene como dirección a memoria un valor aleatorio, por lo que la cadena de caracteres se puede guardar en un momento dado en un área de la memoria insegura. El resultado de la ejecución dependerá del compilador utilizado.

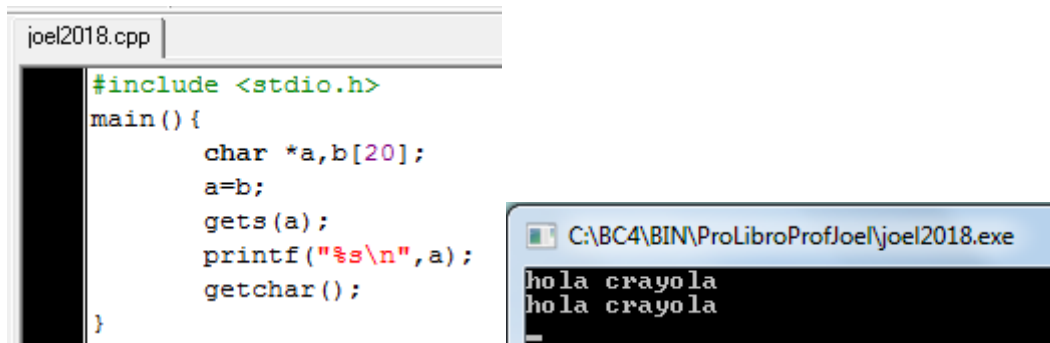
Es importante recordar que una vez que una variable de carácter se le asigne una dirección de memoria, ésta sea una dirección segura.

Al ejecutar el programa 3.18 en DEV-C, se obtiene un desplegado de error después de introducir un texto y en Turbo C se desplegará el mensaje <null>.



Programa 3.18

Observe el programa 3.19:



The image shows a code editor window titled 'joel2018.cpp' containing the following C++ code:

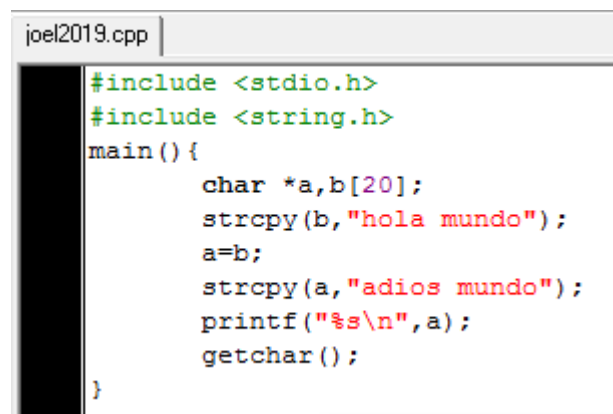
```
#include <stdio.h>
main() {
    char *a,b[20];
    a=b;
    gets(a);
    printf("%s\n",a);
    getchar();
}
```

To the right of the code editor is a screenshot of the program's execution. The title bar of the window reads 'C:\BC4\BIN\ProLibroProfJoel\joel2018.exe'. The console output shows two lines of text: 'hola crayola' and 'hola crayola'.

Programa 3.19

En este programa, la variable `b` acepta hasta 19 caracteres en zona segura. Por lo que el apuntador `a` recibe la dirección de inicio del arreglo `b`. De esta forma se puede leer una cadena de caracteres del teclado en zona segura de estos 19 bytes.

Observe el programa 3.20 e indique que es lo que se despliega en el monitor al momento de su ejecución:



The image shows a code editor window titled 'joel2019.cpp' containing the following C++ code:

```
#include <stdio.h>
#include <string.h>
main() {
    char *a,b[20];
    strcpy(b,"hola mundo");
    a=b;
    strcpy(a,"adios mundo");
    printf("%s\n",a);
    getchar();
}
```

Programa 3.20

Al cuestionarnos ¿Qué muestra en pantalla la ejecución del programa? Si pensó que se muestra en monitor "hola mundo", es un error. Recuerde que al momento de hacer la igualdad `a=b`, la variable `a` tiene como valor el inicio del arreglo donde apunta la variable `b`, por lo que se sobre escribirá la palabra "adiós mundo", siendo esta frase la que se mostrará.

### 3.4 Arreglos de apuntadores.

Un arreglo es una colección de datos homogéneos, y un apuntador es una variable especial que guarda una dirección de memoria. Ahora bien, es posible y práctico contar con un arreglo de apuntadores. Observe el programa 3.21:

```
joel2020.cpp
#include <stdio.h>
#include <string.h>
main() {
    char b[3][20];
    strcpy(b[0], "hola mundo");
    strcpy(b[1], "estoy contento");
    strcpy(b[2], "con la vida");
    printf("%s %u\n", b, b);
    for (int i=0; i<3; i++)
        printf("%s %u\n", b[i], b[i]);
    getchar();
}
```

```
C:\BC4\BIN\ProLibroProfJoel\joel2020.exe
hola mundo 2293504
hola mundo 2293504
estoy contento 2293524
con la vida 2293544
```

Programa 3.21

En este programa, el arreglo bidimensional **b** acepta 3 cadenas con un máximo de 19 caracteres. Observe que el compilador reservará espacio para 20 caracteres (incluyendo el fin de cadena) para cada una de las cadenas. La variable **b** es un apuntador que guarda el inicio del primer elemento, en este caso "b=b[0]", b[0] guarda el inicio de la primer cadena de caracteres, b[1] guarda el inicio de la segunda cadena de caracteres, y b[2] guarda el inicio de la tercera cadena.

Ahora, observe el programa 3.22:

```
#include <stdio.h>
main() {
    char *b[3];
    b[0]="hola mundo"; //La cadena es de 11 bytes incluyendo el byte NULL
    b[1]="estoy contento"; //La cadena es de 15 bytes incluyendo el byte NULL
    b[2]="con la vida";
    for (int i=0; i<3; i++)
        printf("%s %u\n", b[i], b[i]);
    getchar();
}
```

```
C:\TC\BIN\Untitled1.exe
hola mundo 4206592
estoy contento 4206603
con la vida 4206618
-
```

Programa 3.22



En este caso se tiene un arreglo de tres apuntadores a carácter. En la ejecución sólo se guardará espacio para cada una de las cadenas en forma individual, al ejecutarse el programa se obtendrá el siguiente desplegado:

```
hola mundo      4206592
estoy contento  4206603    //Una diferencia de 11 bytes
con la vida     4206618    //Una diferencia de 15 bytes.
```

Como se observó en el ejemplo anterior, existe una diferencia (sutil) entre apuntadores y arreglos. Un ejemplo claro es el manejo de cadenas es:

```
char *p[10];
char mc[10][20];
```

Donde es totalmente valido direccionarse en el lenguaje C a `p[5][6]` y `mc[5][6]`. Sin embargo:

- En el ejemplo anterior, **-mc-** es un arreglo verdadero de 200 elementos de dos dimensiones del tipo char.
- El acceso de cada uno de los elementos del arreglo **-mc-** en memoria se hace bajo la siguiente fórmula:  $20 * \text{renglón} + \text{columna} + \text{dirección base}$ .
- En cambio, la variable **-p-** sólo tiene 10 apuntadores a elementos.
- Una desventaja de la variable **-mc-** es que sólo tiene 200 caracteres disponibles en zona segura.
- Una ventaja de la variable **-p-** es que cada apuntador puede apuntar a cadenas de caracteres de diferente longitud.
- Reiterando, un apuntador es una variable. Si la variable **-ap-** es un apuntador tipo carácter y la variable **-c-** es un arreglo de tipo carácter, se permite realizar las siguientes instrucciones: `"ap=c; ap++;"`. Sin embargo no es posible realizar las siguientes instrucciones: `"c=ap; c++;"`, es ilegal. Recordar que **"un apuntador es una variable, un arreglo NO ES una variable"**.

Por ejemplo, la declaración:

```
char semanam[][15]={"Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"}
```

La variable `semanam` es una matriz con 15 columnas, se puede observar en la figura 3.2.

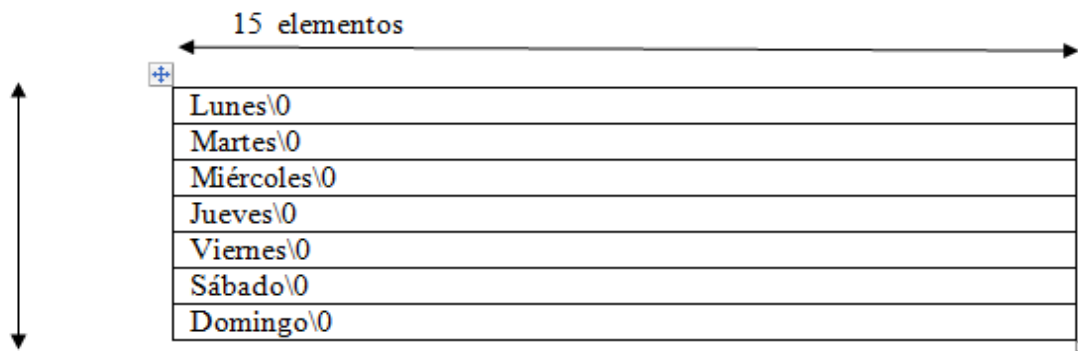


Figura 3.2

El arreglo de punteros semanap:

```
char *semanap[]={“Lunes”, “Martes”, “Miércoles”, “Jueves”, “Viernes”, “Sábado”,
“Domingo”}
```

Se representa gráficamente en la figura 3.3:

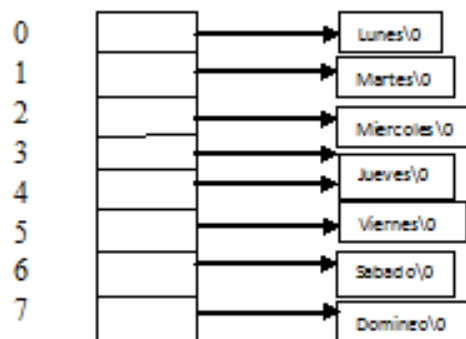


Figura 3.3

Se observa con claridad que en la variable **semanap** tiene un uso más eficiente del espacio.

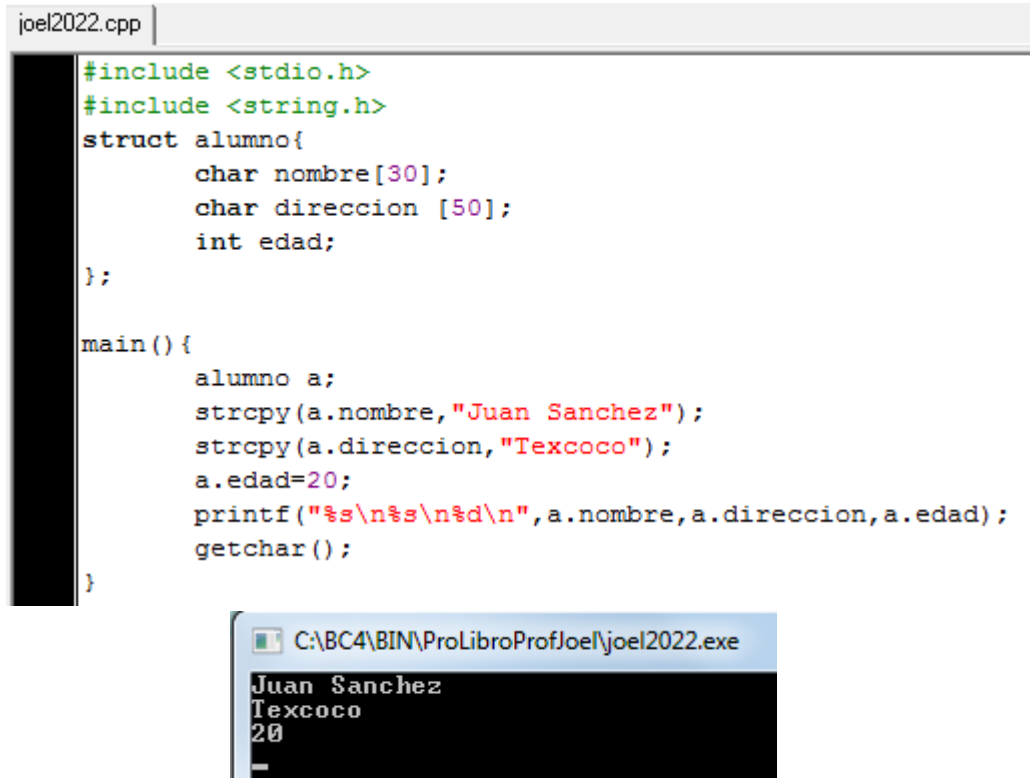
### 3.5 Manejo de Estructuras (registros o tuplas)

Una estructura es una colección de variables de diferentes tipos de datos. Por supuesto, los arreglos permiten un medio de almacenar variables dentro de una unidad. En el caso de los arreglos, los valores dentro del arreglo deben de ser homogéneos y en el caso de las estructuras, los valores son heterogéneos.

Existen diferentes formas de crear una estructura, la más común es:

```
struct alumno{
    char nombre[30];
    char dirección [50];
    char sexo;
    int edad;
};
```

El nombre de la estructura es en sí un tipo de dato. Por lo que se pueden declarar variables tipo nombre de la estructura. Por ejemplo, el programa 3.23 muestra en pantalla los elementos guardados en la estructura –**alumno**–:



The image shows a code editor window titled 'joel2022.cpp' containing the following C++ code:

```
#include <stdio.h>
#include <string.h>
struct alumno{
    char nombre[30];
    char direccion [50];
    int edad;
};

main() {
    alumno a;
    strcpy(a.nombre, "Juan Sanchez");
    strcpy(a.direccion, "Texcoco");
    a.edad=20;
    printf("%s\n%s\n%d\n", a.nombre, a.direccion, a.edad);
    getchar();
}
```

Below the code editor is a screenshot of the program's execution. The window title is 'C:\BC4\BIN\ProLibroProfJoel\joel2022.exe'. The output displayed is:

```
Juan Sanchez
Texcoco
20
```

Programa 3.23

Algunos compiladores permiten la declaración de variable de la siguiente forma:

alumno a;

Otros compiladores requieren que el tipo de variable sea más específico, por lo que se debe de anteponer la palabra struct:

struct alumno a;

La forma de tener acceso a un elemento de la estructura es:

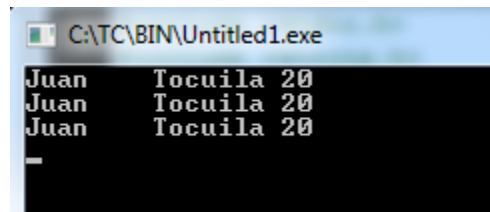
a.nombre.

Los apuntadores a estructuras son muy comunes en la programación en C. Cuando se utiliza un apuntador a estructuras, se requiere un operador especial para poder tener acceso a un elemento de una estructura. El operador es “->” (Un guion medio seguido del signo mayor que). El programa 3.24 muestra el uso de un apuntador tipo estructura:

```

#include <stdio.h>
#include <string.h>
struct alumno{
char a[20];
char b[30];
int edad;
};
main(){
    struct alumno alumnos,*b;
    b=&alumnos;
    strcpy(b->a,"Juan");
    strcpy(b->b,"Tocuila");
    b->edad=20;
    printf("%s\t%s\t%d\n", alumnos.a, alumnos.b, alumnos.edad);
    printf("%s\t%s\t%d\n", b->a, b->b, b->edad);
    printf("%s\t%s\t%d\n", (*b).a, (*b).b, (*b).edad);
    getchar();
}

```



Programa 3.24

En este programa la variable **–alumnos–** es del tipo alumno y la variable **–b–** es un apuntador del tipo alumno. El operador especial “.” se emplea para tener acceso a la estructura al momento de usar la variable **–alumnos–** y el operador especial “->” se emplea para tener acceso a cada uno de los elementos de la estructura al momento de utilizar la variable **–b–**. El último operador es una forma sencilla de acceso a elementos de estructura. Como se observa en el programa, el operador especial “->” tiene la siguiente equivalencia:

$$b \rightarrow edad \equiv (*b).edad$$

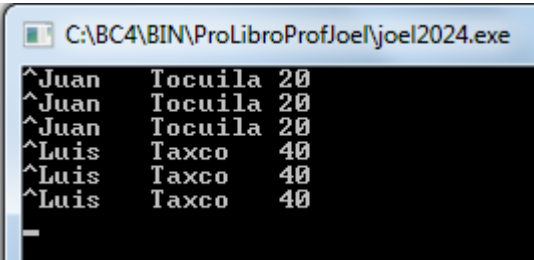
El operador “.” tiene mayor precedencia con respecto al operador unario “\*”, por lo que el operador unario requiere colocarse entre paréntesis.

Como se muestra en el programa 3.25, se pueden tener arreglos de estructuras que se pueden manipular con apuntadores.

```
joel2024.cpp
#include <stdio.h>
#include <string.h>
struct alumno{
    char a[20];
    char b[30];
    int edad;
};

main(){
    struct alumno alumnos[2],*b;
    b=alumnos;    //o también se puede colocar b=&alumnos[0];
    strcpy(b->a, "Juan");
    strcpy(b->b, "Tocuila");
    b->edad=20;
    strcpy((b+1)->a, "Luis");
    strcpy((b+1)->b, "Taxco");
    (b+1)->edad=40;
    printf("^%s\t%s\t%d\n", alumnos[0].a, alumnos[0].b, alumnos[0].edad);
    printf("^%s\t%s\t%d\n", b->a, b->b, b->edad);
    printf("^%s\t%s\t%d\n", (*b).a, (*b).b, (*b).edad);

    printf("^%s\t%s\t%d\n", alumnos[1].a, alumnos[1].b, alumnos[1].edad);
    printf("^%s\t%s\t%d\n", (b+1)->a, (b+1)->b, (b+1)->edad);
    printf("^%s\t%s\t%d\n", (*(b+1)).a, (*(b+1)).b, (*(b+1)).edad);
    getchar();
}
```



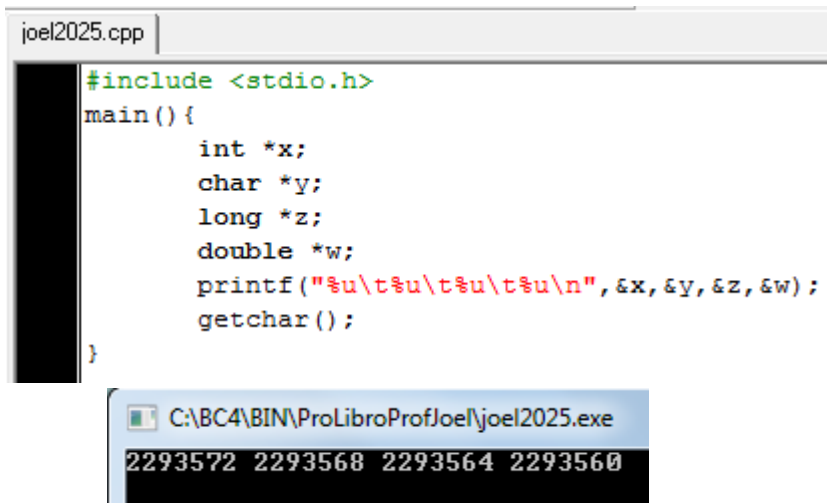
Programa 3.25

En este ejemplo se muestran dos formas diferentes de imprimir el contenido de los registros utilizando los dos operadores especiales para el acceso de los elementos de cada registro. Observe que al incrementar la variable-**b**- en uno, el incremento se realizará con respecto al número de bytes al tamaño de la estructura.

### 3.6 Apuntadores a apuntadores.

Los apuntadores, como cualquier otra variable, también tienen dirección de memoria; esto es, al momento de ser declarados los apuntadores tienen su propia área de memoria. En un modelo pequeño de memoria un apuntador requerirá 2 bytes para el almacenamiento de

una dirección de memoria y para un modelo grande de memoria se requerirán 4 bytes. Se sabe que toda variable tiene dos valores, el primero es la dirección donde se guardará la información, el segundo es el valor del objeto o dato que será guardado en esa dirección. En el caso de apuntadores, el primer valor es la dirección donde se guardará la información del apuntador y el segundo valor es el valor del objeto, pero en este caso, el objeto es otra dirección de memoria a la que está haciendo referencia tal apuntador. Es obvio que los apuntadores tienen una dirección fija donde guarden su objeto. Observemos el programa 3.26:



```
joel2025.cpp
#include <stdio.h>
main() {
    int *x;
    char *y;
    long *z;
    double *w;
    printf("%u\t%u\t%u\t%u\n", &x, &y, &z, &w);
    getch();
}
```

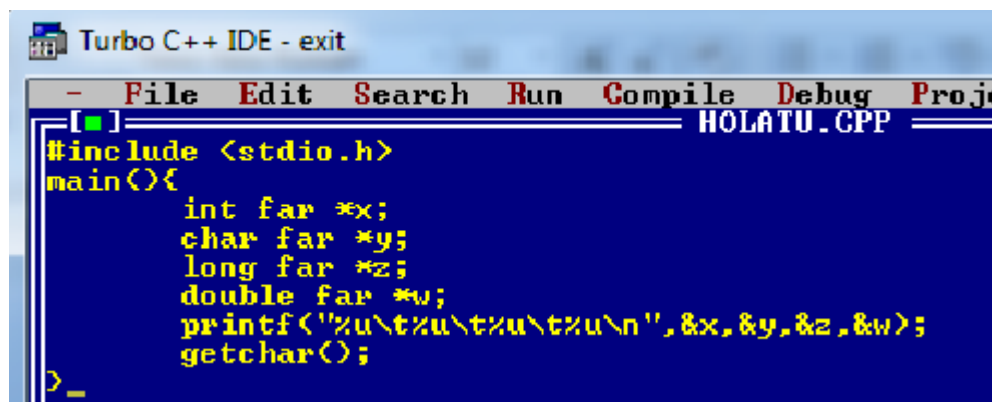
C:\BC4\BIN\ProLibroProfJoel\joel2025.exe

2293572 2293568 2293564 2293560

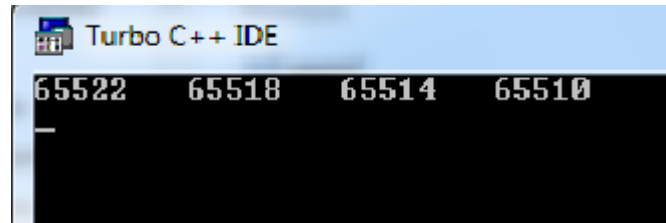
Programa 3.26

Lo que se observa en pantalla es la dirección de memoria donde cada apuntador guardará la dirección de memoria a la que se hace referencia.

Ahora, observe el programa 3.27 ejecutado en TC donde se está manejando un modelo de memoria grande, por lo que se requieren cuatro bytes para manejar la información a guardar:



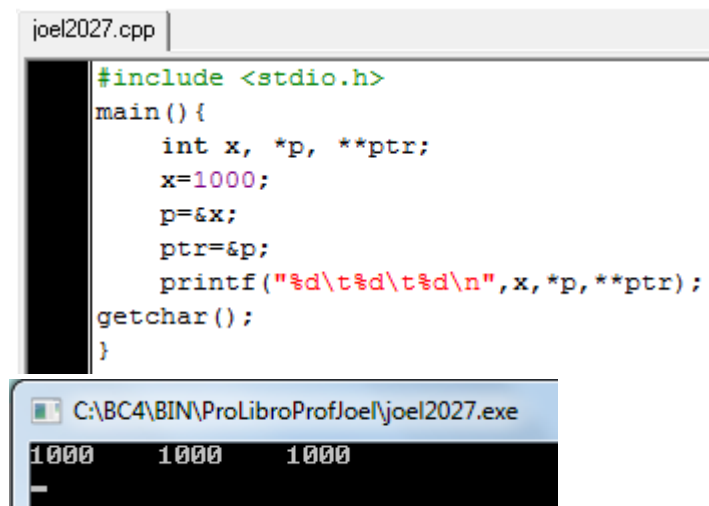
```
Turbo C++ IDE - exit
File Edit Search Run Compile Debug Project
HOLATU.CPP
#include <stdio.h>
main(){
    int far *x;
    char far *y;
    long far *z;
    double far *w;
    printf("%u\t%u\t%u\t%u\n", &x, &y, &z, &w);
    getch();
}
```



```
Turbo C++ IDE
65522  65518  65514  65510
```

Programa 3.27

Se puede tener una variable que apunte a la dirección de memoria de la variable tipo apuntador. El programa 3.28 muestra el uso de un apuntador a apuntador o doble apuntador:



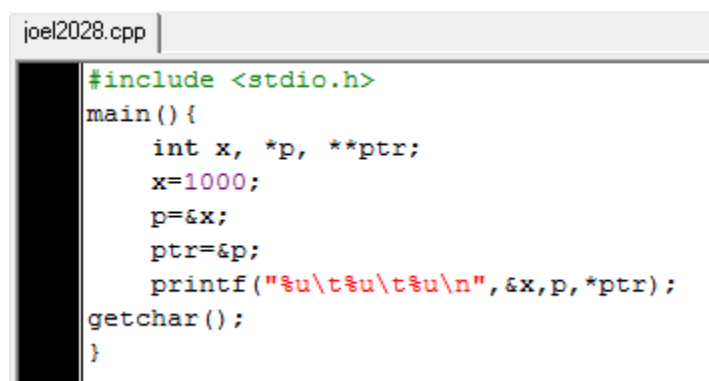
```
joel2027.cpp
#include <stdio.h>
main() {
    int x, *p, **ptr;
    x=1000;
    p=&x;
    ptr=&p;
    printf("%d\t%d\t%d\n",x,*p,**ptr);
    getch();
}

C:\BC4\BIN\ProLibroProfJoel\joel2027.exe
1000  1000  1000
```

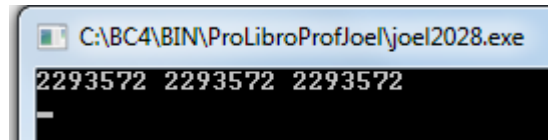
Programa 3.28

En este programa **-\*\*ptr-** es declarado un doble apuntador, o apuntador a apuntador, de tipo entero, por lo que da acceso al objeto que guarda la variable **-p-** y de esta forma accederá al objeto guardado por la variable **-x-**.

Observe el programa 3.29:



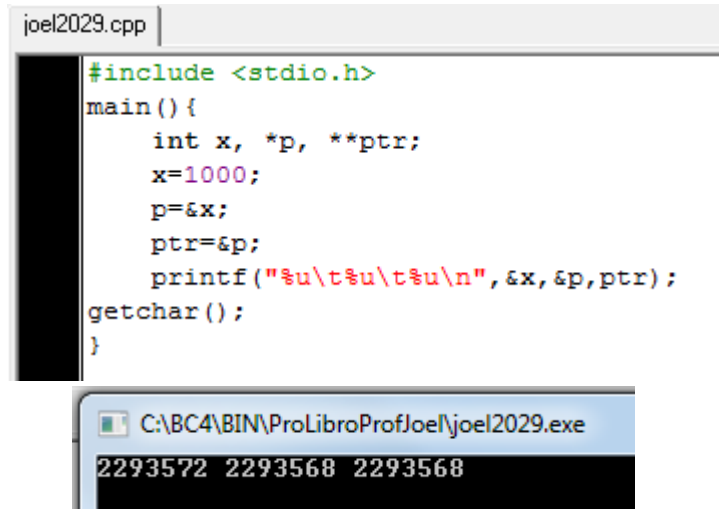
```
joel2028.cpp
#include <stdio.h>
main() {
    int x, *p, **ptr;
    x=1000;
    p=&x;
    ptr=&p;
    printf("%u\t%u\t%u\n",&x,p,*ptr);
    getch();
}
```



Programa 3.29

La variable **-p-** guarda la dirección de la variable **-x-** donde hace referencia, y la variable **-\*ptr-** guarda la dirección donde hace referencia **-p-** (siendo ptr un doble apuntador) lo que se muestra en pantalla es la misma dirección de memoria para las tres variables.

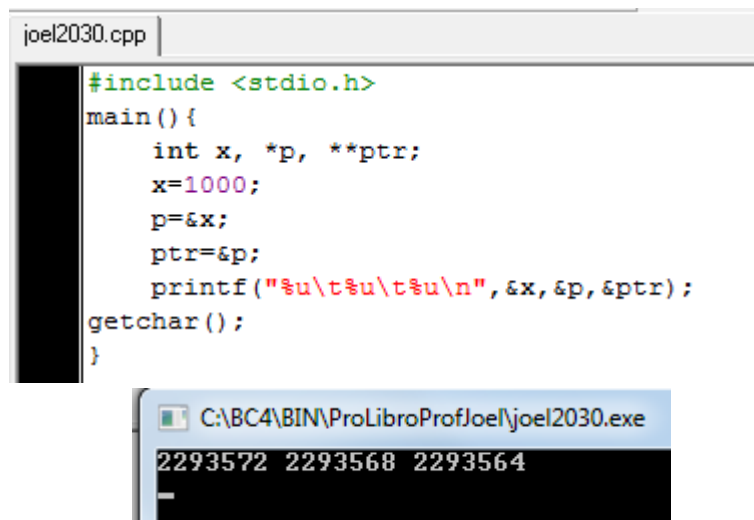
Observe el programa 3.30:



Programa 3.30

Donde la variable **-&p-** muestra la dirección de memoria donde la variable **-p-** guardará el valor del objeto y **-ptr-** hará referencia al valor del objeto guardado en su área de memoria.

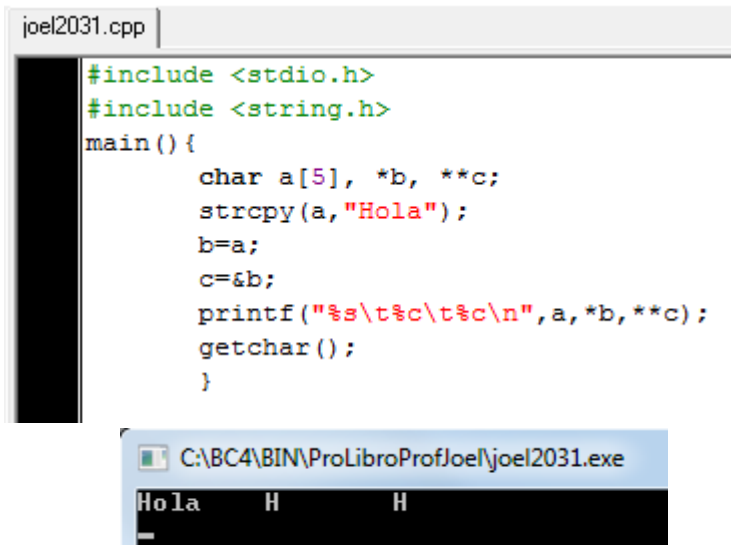
El programa 3.31 muestra el apuntador al doble apuntador:



Programa 3.31



-&ptr- es la dirección de memoria donde -ptr- guarda el valor del objeto.  
El programa 3.32 muestra el uso de apuntadores a apuntadores tipo carácter:



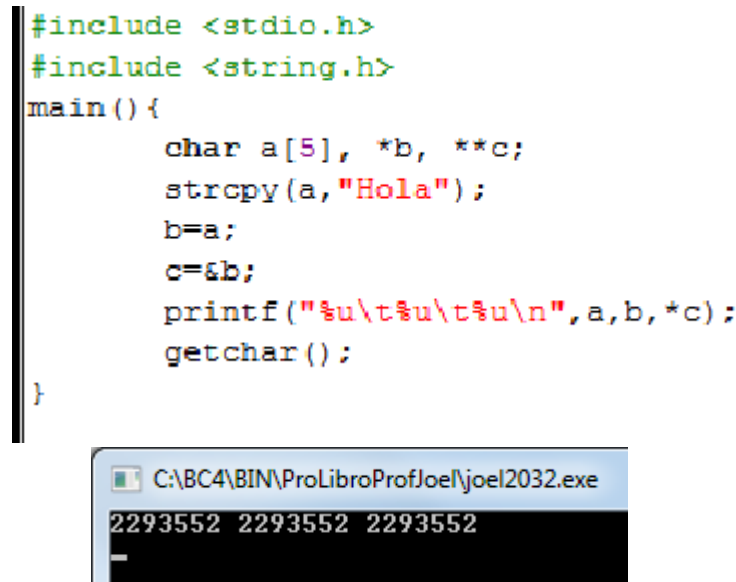
```
joel2031.cpp
#include <stdio.h>
#include <string.h>
main() {
    char a[5], *b, **c;
    strcpy(a, "Hola");
    b=a;
    c=&b;
    printf("%s\t%c\t%c\n", a, *b, **c);
    getchar();
}
```

C:\BC4\BIN\ProLibroProfJoel\joel2031.exe

Hola H H

Programa3.32

El operador unario “\*” se utiliza para diferenciar a los objetos buscados. Un doble apuntador se puede pensar como un apuntador anidado. Observe el programa 3.33:



```
#include <stdio.h>
#include <string.h>
main() {
    char a[5], *b, **c;
    strcpy(a, "Hola");
    b=a;
    c=&b;
    printf("%u\t%c\t%c\n", a, b, *c);
    getchar();
}
```

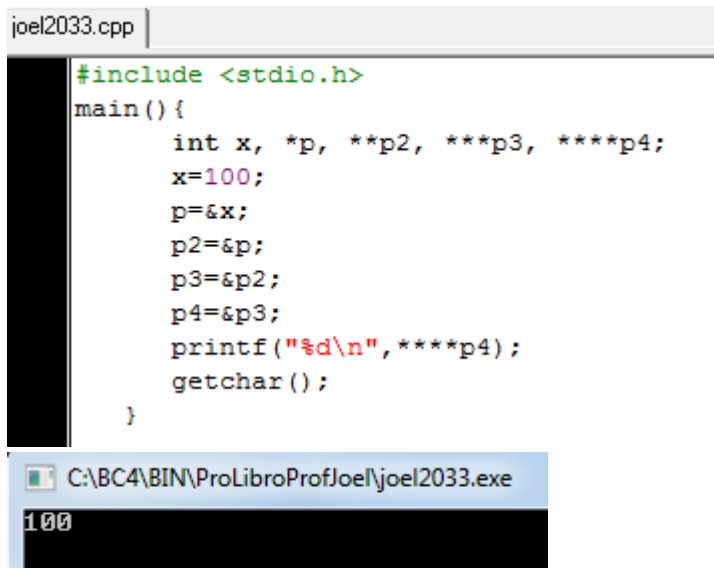
C:\BC4\BIN\ProLibroProfJoel\joel2032.exe

2293552 2293552 2293552

Programa 3.33

En este caso, -\*c- es la dirección de memoria del objeto anidado. La expresión -\*\*c- es el objeto al final de la pila, por lo que se direccionará al primer carácter de la cadena. Observe que si solicita la impresión de una cadena para la variable -c- (%s) en vez de la impresión de un carácter (%c), se producirá un error de ejecución.

En C se puede crear un mayor número de anidamientos. El programa 3.34 muestra un ejemplo de anidamiento:



The image shows a code editor window titled 'joel2033.cpp' containing the following C code:

```
#include <stdio.h>

main() {
    int x, *p, **p2, ***p3, ****p4;
    x=100;
    p=&x;
    p2=&p;
    p3=&p2;
    p4=&p3;
    printf("%d\n", ****p4);
    getchar();
}
```

Below the code editor is a console window titled 'C:\BC4\BIN\ProLibroProfJoel\joel2033.exe' showing the output '100'.

Programa 3.34

Este programa muestra la capacidad de anidamiento de las operaciones con apuntadores, por supuesto, es difícil encontrar en la práctica un anidamiento tan profundo.

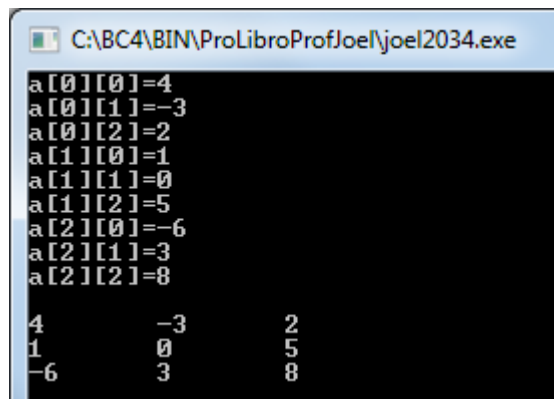
### 3.7 Manejo de matrices con apuntadores.

Una matriz es un arreglo bidimensional de números. Las matrices se usan generalmente para describir sistemas de ecuaciones lineales, sistemas de ecuaciones diferenciales, representar una aplicación lineal o un grafo.

Siendo una matriz un arreglo bidimensional, se puede manipular por medio de corchetes o por anidamiento de apuntadores como se muestra en el programa 3.35:

```
joel2034.cpp
#include <stdio.h>
main() {
    int a[3][3], i, j;
    for (i=0; i<3; i++)
        for (j=0; j<3; j++) {
            printf("a[%d][%d]=", i, j);
            scanf("%d", (*(a+i)+j));
        }
    for (i=0; i<3; i++)
    { printf ("\n");
      for (j=0; j<3; j++)
          printf("%d\t", (*(a+i)+j));

    }
    getchar();
    getchar();
}
```



```
C:\BC4\BIN\ProLibroProfJoel\joel2034.exe
a[0][0]=4
a[0][1]=-3
a[0][2]=2
a[1][0]=1
a[1][1]=0
a[1][2]=5
a[2][0]=-6
a[2][1]=3
a[2][2]=8

4      -3      2
1       0       5
-6      3       8
```

Programa 3.35

Analice los programas anteriores sobre apuntadores dobles y trate de comprender el funcionamiento del último programa.

Ejercicios:

1. En el manejo de memoria dinámica, indique la diferencia entre los siguientes signos: "&" y "\*".
2. Indique la diferencia entre las siguientes declaraciones de variables:
  - a. `int i;`
  - b. `int *pi;`
3. Para el siguiente código indique la diferencia entre `p`, `*p` u `&p`, favor de graficar:

```
Int *p, i=10;  
p=&i;
```

4. Comente el siguiente código:

```
int i=50;  
int *p;  
p=&i;  
*p=100;  
printf("%d", i);
```

5. Explique lo que imprime el siguiente código:

```
char *a;  
a="hola";  
for (int x=0; *(a+x)!='\n';x++)  
    printf("%c", *(a+x));
```

6. Explique lo que imprime el siguiente código:

```
char *a;  
a="hola mundo";  
while (*a)  
    printf("%c", *a++);
```

7. Comente el siguiente código:

```
char *a;  
strcpy(a, "hola mundo");  
printf("%s\n", a);
```

8. Comente el siguiente código:

```
char *b[3];  
b[0]="hola";  
b[1]="mundo";  
b[2]="adiós";  
for (int i=0; i<3;i++)  
    printf("%s\tu\n", b[i], b[i]);
```

9. Comente ventajas y desventajas de cada una de los siguientes códigos:

- a. char \*p[10];
- b. char mc[10][20];

10. Explique lo que es una estructura en el lenguaje de programación C y de un ejemplo.

11. Comente el siguiente código y su impresión:

```
int x, *p, **ptr;  
x=100;  
p=&x;  
ptr=&p;
```

```
printf("%d\t%d\t%d\n", x, *p, **ptr);
```

12. Comente el siguiente código, diagrame e indique lo que imprime:

```
int x, *p, **ptr;
```

```
x=100;
```

```
p=&x;
```

```
ptr=&p;
```

```
printf("u\tu\tu\n", &x, p, *ptr);
```

13. Comente el siguiente código, diagrame e indique lo que imprime:

```
int x, *p, **p2, ***p3, ****p4;
```

```
x=100; p=&x; p2=&p; p3=&p2; p4=&p3;
```

```
printf("%d\n", ****p4);
```

14. Comente el siguiente código, observe que en la función de lectura sólo existe un apuntador y en la función de escritura se tienen dos apuntadores,

```
Int a[3][3],i,j;
```

```
For (i=0;i<3;i++)
```

```
    For (j=0;j<3;j++){
```

```
        Scanf("%d",(*(a+i)+j));
```

```
        Printf("%d..",(*(a+i)+j));
```

```
    }
```

## CUARTO CAPITULO: ALLOCATE. Asignar memoria en forma dinámica.

### Resumen.

Este capítulo forma parte de la primera unidad de competencia dentro de “Variables dinámicas: Apuntadores, Operaciones básicas” y muestra cómo se crea una variable en forma dinámica (en el momento de la ejecución y no en el momento de la compilación). Se utiliza el lenguaje de programación C estándar porque permite comprender: primero, el tamaño de memoria a utilizar (dependiendo del Tipo de Dato Abstracto); segundo, la asignación de memoria por medio de una función específica localizada en la librería “stdlib.h” o “alloc.h” (allocate (asignar), para los amantes de turbo c); y tercero, el poder hacer el moldeo o casting. Todos estos elementos son, para el autor, fundamentales que se entiendan ya que esto permite comprender mucho mejor la creación de espacio en memoria principal de un tipo de dato abstracto manejado en forma dinámica en el momento de ejecución.

Los programas son de propia autoría y las ideas básicas se tomaron de los siguientes libros:

(Backman, 2012)

(Ceballos, 2015)

(Deitel & Deitel, 1995)

### 4.1 Apuntadores y asignación de memoria.

En la mayoría de los programas explicados en los capítulos anteriores, los apuntadores se asignaron a localidades de memoria correspondientes a una variable en particular. (Algunos lo conocen como peek/poke (peek se refiere a leer el contenido de una celda de memoria de una determinada dirección y poke establece el contenido de una celda de memoria). Los apuntadores se asignaron a localidades de memoria específicamente con el propósito de leer el contenido de una variable o escribir el contenido de una variable en un área de memoria específica. Estas localidades de memoria se conocen en tiempo de compilación (También conocidas como variables estáticas ya que al compilar se conoce la cantidad de memoria a ser utilizada, por ejemplo: `char a[10]`)

El lenguaje de programación C permite crear localidades de memoria en forma dinámica, es decir, se puede solicitar la asignación de memoria en el momento de ejecución en vez del momento de compilación. Para lograr esto, el lenguaje de programación C tiene las siguientes herramientas:

**stdlib.h** (*std-lib*: *standard library* o biblioteca estándar) es el archivo de cabecera de la biblioteca estándar de propósito general del lenguaje de programación C. Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras. Es compatible con C++ donde se conoce como `cstdlib`.

**alloc.h** (*allocate library* o biblioteca para asignación dinámica de memoria) es el archivo de cabecera para C que nos da la capacidad de asignar y liberar memoria de forma dinámica. No todos los compiladores tienen esta biblioteca, por lo que se debe de verificar si el compilador a utilizar contiene tal herramienta.

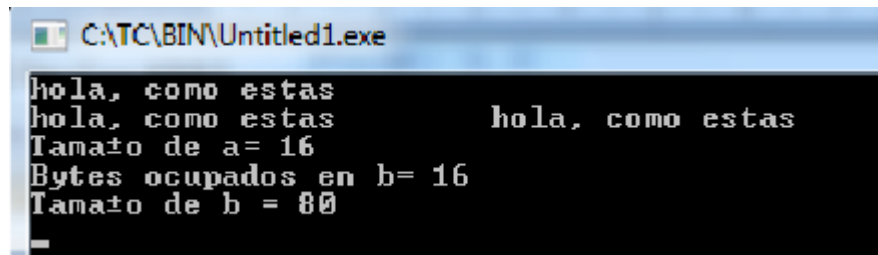
Ambas bibliotecas tienen varias funciones en común, las que se utilizarán en este capítulo son (ver tabla 4.1):

Tabla 4.1 (Plauger, 1992)

Gestión de memoria dinámica	
malloc	<p><code>void * malloc(size_t size)</code></p> <p>La función malloc reserva espacio en el heap a un objeto cuyo tamaño es especificado en bytes (size). El heap es un área de memoria que se utiliza para guardar estructuras dinámicas en tiempo de ejecución. Si se puede realizar la asignación de memoria, se retorna el valor de la dirección de inicio del bloque. De lo contrario, si no existe el espacio requerido para el nuevo bloque, o si el tamaño es cero, retorna NULL.</p>
calloc	<p><code>void *calloc(size_t nitems, size_t size)</code></p> <p>La función calloc asigna espacio en el heap de un número de elementos (nitems) de un tamaño determinado (size). El bloque asignado se limpia con ceros. Si se desea asignar un bloque mayor a 64kb, se debe utilizar la función farcalloc.</p> <p>Ejemplo:  <code>char *s=NULL;</code>  <code>str=(char*) calloc(10,sizeof(char));</code></p> <p>Si se puede realizar la asignación, se retorna el valor del inicio del block asignado. De lo contrario, o no existe el espacio requerido para el nuevo bloque, o el tamaño es cero, retorna NULL.</p>
realloc	<p><code>void *realloc(<i>punt, nuevotama</i>):</code> Cambia el tamaño del bloque apuntado por <i>punt</i>. El nuevo tamaño puede ser mayor o menor y no se pierde la información que hubiera almacenada (si cambia de ubicación se copia).</p>
free	<p><code>void free(void *dir_memoria)</code></p> <p>La función free libera un bloque de la memoria del heap. El argumento <i>dir_memoria</i> apunta a un bloque de memoria previamente asignado a través de una llamada a calloc, malloc o realloc. Después de la llamada, el bloque liberado estará disponible para asignación.</p>

El programa 4.1 muestra el uso de la función –malloc()–:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main() {
    char *a, b[80];
    gets(b);
    a=(char *)malloc(strlen(b));
    strcpy(a,b);
    printf("%s\t%s\n",a,b);
    printf("Tamaño de a= %d\n", strlen(a));
    printf("Bytes ocupados en b= %d\n",strlen(b));
    printf("Tamaño de b = %d\n",sizeof(b));
    getchar();
}
```



Programa 4.1

En este programa se pueden escribir hasta 80 caracteres, incluyendo el null, sin invadir espacio de otras variables. La función –strlen()– pertenece al archivo de cabecera –string.h– y calcula el tamaño de una cadena sin incluir el carácter NULL. La función –sizeof()– no requiere algún archivo de cabecera y retorna el número de bytes ocupados en la memoria por una variable. Cuando se ejecuta la siguiente línea:

```
a=(char *)malloc(strlen(b));
```

la mayor prioridad la tiene la función –strlen()–, por lo que retorna el número de bytes ocupados en la variable –b–, observe que no se utilizó la función –sizeof()– ya que esta función retorna el tamaño de bytes reservados en la variable –b– (en este caso sería 80). Una vez determinado el tamaño, la función –malloc()– reserva el espacio en bytes en el heap, posteriormente se realiza el casting (se crea el espacio de memoria a lo indicado dentro del paréntesis, en este caso, se devuelve un apuntador a carácter al inicio de la cadena). El casting es necesario ya que la función –malloc()– retorna un apuntador tipo void (un apuntador sin tipo). Cuando es un apuntador tipo void, se puede tener la dirección de cualquier tipo de variable pero sin tener el permiso de acceso a ella para escribir o para leer.

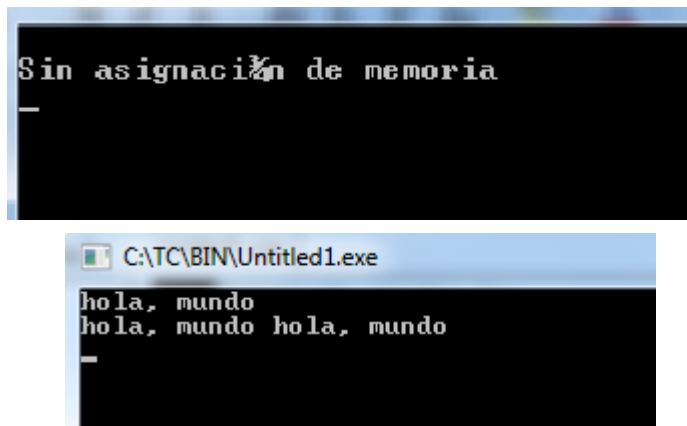


Si se requiere tener el acceso para leer o escribir, se requiere moldear al tipo de variable al que se quiere tener acceso (o casting).

Por lo tanto, al utilizar una variable tipo apuntador y crear memoria en forma dinámica nos permite utilizar sólo la memoria necesaria y no se tiene desperdicio de memoria como se observa con un arreglo tipo cadena de caracteres.

En el programa 4.2 se muestra un ejemplo de la forma de detectar el retorno de NULL en la función malloc(). Si al ejecutar el programa, el ejecutor del programa oprime la tecla return (enter), se mostrará en monitor "Sin asignación de memoria". De otra forma se mostrará dos veces lo escrito por el ejecutor del programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main() {
    char *a, b[80];
    gets(b);
    if ((a=(char *)malloc(strlen(b)))==NULL) {
        printf("Sin asignación de memoria\n");
        getchar();
        exit(0);
    }
    strcpy(a,b);
    printf("%s %s\n",a,b);
    getchar();
}
```



Programa 4.2

Uno puede tener un apuntador que controle a un gran bloque de memoria para algún propósito en específico. El programa 4.3 declara un apuntador entero y utiliza la función malloc para retornar un gran bloque de enteros:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main() {
    int *x, i;
    if ((x=(int *)malloc(200))==NULL) {
        printf("No se asignó la memoria solicitada\n");
        getchar();
        exit(0);
    }
    for (i=0;i<100;i++)
        *(x+i)=i
        ;
    for (i=0;i<100;i++)
        printf("%d..", *(x+i));
    getchar();
    free(x);
}

```

Programa 4.3

Observe algo interesante, malloc(200) asignará memoria para 200 bytes y, al momento de realizar el casting, serán tratados como enteros. (Recuerde que Turbo C, un entero se almacena en dos bytes, por lo que sólo se permite guardar en zona segura a 100 números enteros). En consecuencia, los apuntadores `-*x`-, `-(x+1)`-, `-(x+2)`- lograron tener acceso a sus respectivas unidades de almacenamiento, donde cada unidad de almacenamiento contiene dos bytes. (Cada tipo de variable tendrá su propia unidad de almacenamiento. Por ejemplo, en TC, las variables tipo char tienen un byte como unidad de almacenamiento y la variable double tiene como unidad de almacenamiento a 8 bytes, etc. El tamaño de bytes de cada tipo de variable depende del tipo de máquina y del compilador)

En el programa anterior, el manejo del bloque de bytes se puede realizar sin problema con los subíndices, esto es:

`*x`  $\equiv$  `x[0]`

`*(x+1)`  $\equiv$  `x[1]`

Para saber la dirección de memoria de un arreglo se puede utilizar:

`x`  $\equiv$  `&x[0]`

$x + 1 \equiv \&x[1]$

El programa 4.4 muestra la asignación de memoria en forma dinámica:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int *a,i;
    if ((a=(int *)malloc(40))==NULL){
        printf("Memoria no asignada\n");
        getchar();
        exit(0);
    }
    for (i=0;i<10;i++){
        a[i]=100+i;
    }
    for (i=10; i<20; i++){
        *(a+i)=100+2*i;
    }
    printf("La variable -a- apunta a la localidad de memoria %u %u\n",a, &a[0]);
    for (i=0;i<20;i++){
        printf("a[%d]=%d\t*(a+%d)=%d\tLoc %u\tLoc %u\n", i, a[i],i,*(a+i),&a[i], a+i);
    }
    getchar();
}
```

```
La variable -a- apunta a la localidad de memoria 8523744 8523744
a[0]=100      *(a+0)=100      Loc 8523744      Loc 8523744
a[1]=101      *(a+1)=101      Loc 8523748      Loc 8523748
a[2]=102      *(a+2)=102      Loc 8523752      Loc 8523752
a[3]=103      *(a+3)=103      Loc 8523756      Loc 8523756
a[4]=104      *(a+4)=104      Loc 8523760      Loc 8523760
a[5]=105      *(a+5)=105      Loc 8523764      Loc 8523764
a[6]=106      *(a+6)=106      Loc 8523768      Loc 8523768
a[7]=107      *(a+7)=107      Loc 8523772      Loc 8523772
a[8]=108      *(a+8)=108      Loc 8523776      Loc 8523776
a[9]=109      *(a+9)=109      Loc 8523780      Loc 8523780
a[10]=120     *(a+10)=120     Loc 8523784      Loc 8523784
a[11]=122     *(a+11)=122     Loc 8523788      Loc 8523788
a[12]=124     *(a+12)=124     Loc 8523792      Loc 8523792
a[13]=126     *(a+13)=126     Loc 8523796      Loc 8523796
a[14]=128     *(a+14)=128     Loc 8523800      Loc 8523800
a[15]=130     *(a+15)=130     Loc 8523804      Loc 8523804
a[16]=132     *(a+16)=132     Loc 8523808      Loc 8523808
a[17]=134     *(a+17)=134     Loc 8523812      Loc 8523812
a[18]=136     *(a+18)=136     Loc 8523816      Loc 8523816
a[19]=138     *(a+19)=138     Loc 8523820      Loc 8523820
```

Programa 4.4

La función -calloc()-, a comparación de la función -malloc()-, limpia la memoria. Esto significa que todo block asignado de memoria será inicializado en cero. El programa 4.5 es similar a un ejemplo anterior en este capítulo y va a permitir hacer la comparación entre las dos funciones.

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int *x,y;
    if((x=(int *)calloc(400,2))==NULL){
        printf("No asignación de memoria");
        exit(0);
    }
}
```

```

    for (y=0; y<400; y++)
        *(x+y)=88;
}

```

#### Programa 4.5

Cuando se invocó a `-malloc(400)-`, se asignó una dirección con un block de 400 bytes a un apuntador tipo `int`. Sabiendo que un entero necesita 2 bytes, se tiene memoria para 200 números enteros. En el programa anterior `-calloc()-` se emplea de la siguiente forma:

`calloc(unidades requeridas, tamaño por unidad)`

Al invocarse `-calloc(400,2)-` indica que se requieren 400 unidades de dos bytes cada una, esto nos da un block de almacenamiento de 800 bytes.

En este caso es importante si se requiere portabilidad de uno a otro tipo de máquina. Como C no es completamente estándar, como lo es Java, existe diferencia en el número de bytes por tipo de variable. Por ejemplo, para una microcomputadora una variable tipo entero se pueden requerir 2 bytes, pero para una mainframe se pueden requerir 4 bytes. La asignación de memoria para todo tipo de dato en C es relativa y se basa más en la arquitectura de la máquina y otros factores.

Una forma simple de verificar el tamaño de bytes requerido para un tipo de variable en un compilador para una arquitectura específica es utilizando la función `-sizeof()-`. Si se espera portar el programa a otra arquitectura se puede realizar lo siguiente:

```

calloc(400, sizeof(int));

```

De esta forma, `calloc()` permite la portabilidad sin ningún problema, o puede utilizarse `malloc()` de la siguiente forma:

```

malloc(400*sizeof(int))

```

En este caso, la única diferencia es que `-malloc()-` no inicializa el bloque asignado de memoria.

Para liberar un bloque de memoria creado por `-malloc()-` o `-calloc()-` se puede usar la función `free()` de la siguiente forma:

```

free(ptr).

```

Donde `-ptr-` es el apuntador al inicio del block de memoria asignado.

La función `-realloc(ptr, tamaño)-` cambia el tamaño del objeto apuntado por `-ptr-` al tamaño especificado por `-tamaño-`. El contenido del objeto no cambiará hasta que se defina el nuevo tamaño del objeto. Si el nuevo tamaño es mayor, el valor de la porción nueva se adjudicará al objeto. Si `-ptr-` es un puntero nulo, la función `-realloc()-` se comporta igual que la función `-malloc()-`. De lo contrario, si `-ptr-` no es igual a un puntero previamente retornado por la

función `–malloc()`-, `–calloc()`- o `–realloc()`, o si el espacio ha sido desadjudicado por una llamada a la función `–free()`- o `–realloc()`, el comportamiento no estará definido.

Por ejemplo, el programa 4.6 lee y escribe los valores de un arreglo `–V–` de reales. El número de valores se conoce durante la ejecución. La memoria se asignará en el momento de ejecución elemento por elemento.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h> /* para getch() */
main() {
float *V=NULL; int N=0,i; char c;
do {
    V=(float *)realloc((float *)V,(N+1)*sizeof(float));
    printf("\nDame valor >>"); scanf("%f",&V[N]);
    printf("Quieres introducir otro valor? (S/N) >> ");
    c=getch();
    N++;
} while (c=='s' || c=='S');
for(i=0;i<N;i++) printf("\nValor %d >> %f\n",i,V[i]);
free(V);
}
```

Programa 4.6

## 4.2 Manejo de matrices en forma dinámica

El lenguaje de programación C proporciona la posibilidad de manejar matrices en forma estática por lo que se debe conocer el tamaño de la matriz en tiempo de compilación. En caso de matrices de dos dimensiones, el compilador debe conocer el número de filas y columnas. Si el tamaño de la matriz se conoce hasta el tiempo de ejecución el lenguaje de programación C nos permite manipular matrices en forma dinámica.

En particular, se posibilitará:

- Crear matrices en forma dinámica (en el momento de ejecución)
- Destruir matrices en forma dinámica.
- Acceso mediante índices o apuntadores.

Para poder trabajar la matriz en forma dinámica se requiere una variable del tipo doble apuntador. El primer apuntador hará referencia al inicio de un arreglo de apuntadores y cada apuntador del arreglo de apuntadores tendrá la referencia del inicio de un arreglo de enteros u otro tipo de variable, ver figura 4.1:

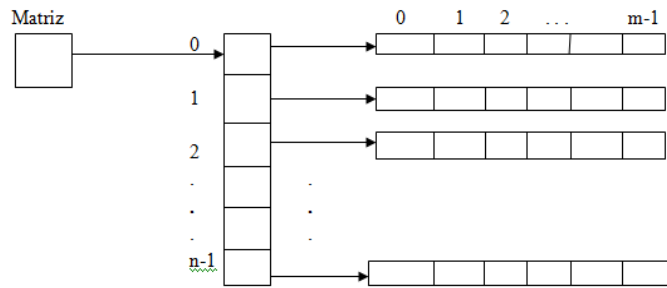


Figura 4.1

El programa 4.7 muestra la forma que debe ser manipulado cada uno de los apuntadores para la creación dinámica de la matriz bidimensional:

```
#include <stdio.h>
#include <stdlib.h>
main(){
    int f,c,i,j;
    int **pm;
    printf("Da el numero de hileras=>");
    scanf("%d",&f);
    getchar();
    printf("Da el numero de columnas=>");
    scanf("%d",&c);
    pm=(int **)malloc(sizeof(int *)*f);
    for (j=0;j<c;j++)
        pm[j]=(int*)malloc(sizeof(int)*c);
    for (i=0;i<f;i++)
        for (j=0;j<c;j++)
            pm[i][j]=i*j+1;
    printf("Mostrando la matriz utilizando corchetes\n");
    for (i=0;i<f;i++){
        for (j=0;j<c;j++)
            printf("%d\t",pm[i][j]);
        putchar('\n');
    }
    printf("Mostrando la matriz utilizando apuntadores\n");
    for (i=0;i<f;i++){
        for (j=0;j<c;j++)
            printf("%d\t",*(pm+i+j));
        putchar('\n');
    }
    getchar();
    getchar();
}
```

Programa 4.7

La siguiente línea del código anterior crea el arreglo de apuntadores:

```
pm=(int **)malloc(sizeof(int *)*f);
```

Donde la función `-sizeof ()-` entrega el número de bytes requerido para un apuntador tipo entero, al multiplicarse por el número de hileras, lo que se tiene es el tamaño real de arreglo de apuntadores para que la función `-malloc()-` realice el apartado de memoria con la magnitud indicada. Al final se realiza un casting para poderse entregar la dirección inicial del arreglo de apuntadores a la variable del tipo doble apuntador a entero.

Para poderse crear cada uno de los arreglos de enteros, se requerirá una instrucción cíclica como es el `"for"` ya que se le buscará espacio en forma independiente a cada arreglo de enteros. La función `-sizeof()-` retornará el tamaño requerido por cada entero, al multiplicarse por el número de columnas se tendrá el tamaño total del arreglo de enteros. La función `-malloc()-` buscará espacio en el HEAP para el tamaño solicitado por la función `-sizeof()-`, al final se realiza un casting tipo apuntador entero. El apuntador que hace referencia al inicio del arreglo se entrega a la variable tipo apuntador a entero `-pm[j]-`.

```
for (j=0;j<c;j++)
    pm[j]=(int*)malloc(sizeof(int)*c);
```

#### 4.3 Manejo de arreglos de registros en forma dinámica.

El programa 4.8 muestra una forma de crear un arreglo de registros en forma dinámica:

```
#include <stdio.h>
#include <stdlib.h>
struct alumno{
    char a[20];
    char b[30];
    int edad;
};
main(){
    alumno *b;
    int max,i;
    printf("Da el numero de registros a manejar=>");
    scanf("%d",&max);
    b=(alumno *)malloc(sizeof(alumno)*max);
    for (i=0;i<max;i++){
        printf("Da el nombre del elemento %d=>",i);
        scanf("%s",(b+i)->a);
        printf("Da la dirección del elemento %d=>",i);
```

```

        scanf("%s",(*b+i).b);
        printf("Da la edad del elemento %d=>",i);
        scanf("%d",&(b+i)->edad);
    }
    for (i=0;i<max;i++){
        printf("%s\t%s\t%d\n",b+i->a,b+i->b,b+i->edad);
        printf("%s\t%s\t%d\n",(*b+i).a,(*b+i).b,(*b+i).edad);
    }
    getchar();
    getchar();
}

```

### Programa 4.8

En este programa, la siguiente línea crea el arreglo de registros:

```
b=(alumno *)malloc(sizeof(alumno)*max);
```

donde la función `-sizeof()-` entrega el número de bytes requerido para un registro tipo `alumno`, al multiplicarse por el número de registros requerido (`max`), lo que se tiene es el tamaño real de arreglo de registros para que la función `-malloc()-` realice el apartado de memoria con la magnitud indicada. Al final se realiza un casting para poderse entregar la dirección inicial del arreglo de registros a la variable del tipo `apuntador de registros`.

El programa muestra cómo realizar la lectura de cadenas de caracteres empleando los dos operadores especiales `"->"` o `"."`. Para poder hacerse la lectura de la edad, siendo ésta del tipo entero, se requiere forzosamente el operador `"&"`. En la lectura de las variables tipo arreglo de carácter no se requiere el operador `"&"` ya que el nombre de la variable es el apuntador al primer elemento de la cadena.

Nota: El operador `-new()-` se puede emplear en lugar del operador `-malloc()-` y el operador `-delete()-` se puede emplear en lugar del operador `-free()-`. Si se decide emplear el operador `-new()-`, para liberar memoria se tiene que utilizar forzosamente el operador `-delete()-`. Si se utiliza el operador `-malloc()-`, para liberar memoria se tiene que utilizar forzosamente el operador `-free()-`. **No es recomendable mezclar operadores.**



## Ejercicios.

1. Comente que es el HEAP.
2. Comente la función "malloc()".
3. Comente la función "free()".
4. Explique la función "strlen()" y compárela con la función "sizeof()".
5. Comente el siguiente código.  

```
a=(char *) malloc (strlen(b));
```
6. Indique el objetivo de la función "calloc()".
7. Indique el objetivo de la función "realloc()".
8. Indique como funciona una matriz en forma dinámica y diagrame.
9. Escriba el código para el manejo dinámico de una matriz de NXM.
10. Escriba el código para el manejo dinámico de un arreglo de registros.
11. Escriba un programa que cree un arreglo de registros en forma dinámica.

## QUINTO CAPITULO: FUNCIONES.

### Resumen.

Este capítulo forma parte de la primera unidad de competencia dentro de “Variables dinámicas: Apuntadores, Operaciones básicas” y explica paso a paso el manejo de funciones, el pase de parámetros por valor y la forma en que maneja el pase de parámetros por referencia, aparte de explicar el retorno de valores. Es fundamental este capítulo ya que muestra la forma en que se pueden hacer módulos más sencillos que permitan un mejor diseño de algún programa en particular. El buen diseño de un módulo se explica en el curso “Programación Avanzada” por medio de los conceptos “cohesión y acoplamiento”.

Los programas son de propia autoría y las ideas básicas se tomaron de los siguientes libros:

(Ceballos, 2015)

(Deitel & Deitel, 1995)

(Backman, 2012)

### 5.1 Paso de parámetros

Las ligaduras de los parámetros tiene un efecto significativo en la semántica de las llamadas a procedimientos, los lenguajes difieren de manera sustancial de las clases de mecanismos como: paso de parámetros disponibles y el rango de los efectos permisibles de implementación que pudieran ocurrir. Algunos lenguajes ofrecen sólo una clase básica de mecanismo de paso de parámetros, mientras que otros pueden ofrecer dos o más. Los pases de parámetros más conocidos son: Paso de parámetros por valor y paso de parámetros por referencia.

**Paso de parámetros por valor.** El paso de parámetros por valor consiste en copiar el contenido de la variable que queremos pasar al ámbito local de la función llamada, consiste en copiar el contenido de la memoria del argumento que se quiere pasar a otra dirección de memoria, correspondiente al argumento dentro del ámbito de dicha función. Se tendrán dos valores duplicados e independientes, con lo que la modificación de uno no afecta al otro.

**Paso de parámetros por referencia.** El paso de parámetros por referencia consiste en proporcionar a la función llamada a la que se le quiere pasar el argumento la dirección de memoria del dato. Con este mecanismo un argumento debe ser en principio una variable con una dirección asignada. En vez de pasar el valor de la variable, el paso por referencia pasa la ubicación de la variable de modo que el parámetro se convierte en un **alias** para el argumento, y cualquier cambio que se le haga a éste lo sufre también el argumento. Los lenguajes que permiten el paso de parámetros se define utilizando sintaxis adicional. Por ejemplo, en Pascal:

```
procedure inc(var x : integer);  
begin
```

```

    x := x+1;
end;

```

En este ejemplo, la variable `-x-` es sólo un alias. Pascal incluye como sintaxis adicional la palabra clave `-var-` indicando que el paso de parámetro es por referencia.

Note que el paso de parámetros por valor no implica que no puedan ocurrir cambios fuera del procedimiento mediante el uso de parámetros. Si el parámetro tiene un tipo de apuntador o referencia, entonces el valor es una dirección, y puede utilizarse para cambiar la memoria por fuera del procedimiento. Por ejemplo, la siguiente función en C cambia de manera definitiva el valor del entero a la cual el parámetro `-p-` apunta:

```

void int_p(int* p){
    *p=0;
}

```

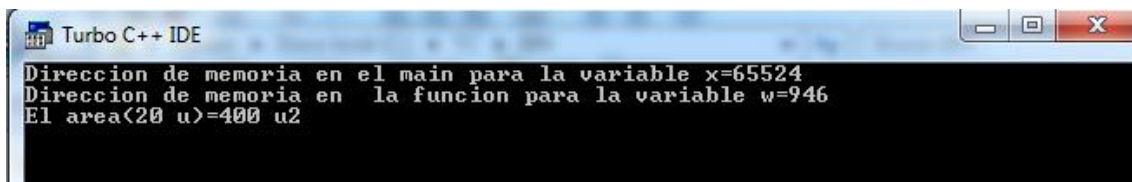
En el lenguaje C, los argumentos de los tipos de variables escalares son pasados por valor. No es posible cambiar el valor de una función llamadora dentro de una función llamada que esté en ejecución, al menos que se tenga la dirección de la variable de la función llamadora que se desee modificar.

Observemos el programa 5.1:

```

#include <stdio.h>
int area_cuadrado(int);
main(){
    int x,y;
    x=20;
    printf("Direccion de memoria en el main para la variable x=%u\n",&x);
    y=area_cuadrado(x); //función llamadora
    printf("El area(%d u)=%d u2\n",x,y);
    getchar();
}
int area_cuadrado(int w){ //función llamada
    printf("Direccion de memoria en la funcion para la variable w=%u\n");
    return(w*w);
}

```



```

Turbo C++ IDE
Direccion de memoria en el main para la variable x=65524
Direccion de memoria en la funcion para la variable w=946
El area(20 u)=400 u2

```

Programa 5.1

Este programa contiene dos funciones: `-main()-` y `-area_cuadrado()-`, cada función tiene sus propias variables locales. Una variable local es aquella cuyo ámbito se restringe a la función que la ha declarado, esto implica que la variable local sólo va a poder ser manipulada en dicha sección, y no se podrá hacer referencia fuera de dicha sección. La función `main()` tiene las variables locales enteras `-x-` y `-y-`. La función `area_cuadrado()` tiene la variable local entera `-w-`.

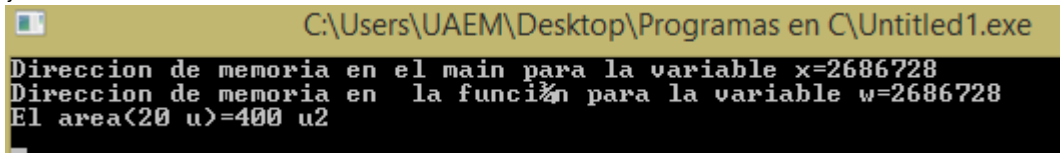
La función `area_cuadrado()` es invocada dentro de la función `main()` (la función llamadora) y el paso de parámetro es por valor. Esto es, `w=x` (El valor contenido en la variable `-x-` será dado a la variable `-w-`).

Observe que al ejecutarse el programa 5.1, la variable `-x-` tiene una dirección de memoria independiente a la variable `-w-` y la variable `-w-` sólo recibe una copia del valor guardado en la localidad de la variable `-x-`. (Observe que la función `area_cuadrado()`, dentro de su declaración indica que va a retornar un valor del tipo entero).

¿Qué sucede si en vez de enviar el valor de la variable se envía su dirección?

Observe el programa 5.2:

```
#include <stdio.h>
void area_cuadrado(int *);
main(){
    int x,y;
    y=x=20;
    printf("Direccion de memoria en el main para la variable x=%u\n",&x);
    area_cuadrado(&x); // función llamadora
    printf("El area(%d u)=%d u2\n",y,x);
    getchar();
}
void area_cuadrado(int *w){ // función llamada
    printf("Direccion de memoria a la que apunta la variable w=%u\n");
    *w=*w**w;
}
```



```
C:\Users\UAEM\Desktop\Programas en C\Untitled1.exe
Direccion de memoria en el main para la variable x=2686728
Direccion de memoria en la función para la variable w=2686728
El area(20 u)=400 u2
```

Programa 5.2

En el programa 5.2, como en el programa anterior, la función `main()` tiene las variables locales enteras `x` y `y`. La función `area_cuadrado()` tiene la variable local apuntador a entero `w`.

A diferencia del programa anterior, en este programa se envía como parámetro de la función llamadora a la dirección de memoria de la variable `x`, por lo que se tiene como argumento en la función llamada a un apuntador a la dirección de memoria de la variable que se utilizó como argumento en la función llamadora. Esto es, `w=&x` (Recuerde que el lenguaje de programación C, para tipos de variables escalares, sólo acepta el pase de parámetros por valor siendo diferente el caso para otro tipo de variables). Esto es como darle permiso al argumento de la función llamada a que modifique el dato guardado en la variable que se utilizó como argumento en la función llamadora (un alias). Esta es la forma en que el lenguaje de programación C simula el paso de parámetros por referencia para los tipos de variables escalares.

## 5.2 Paso de parámetros en vectores.

Como recordaremos, al declarar el nombre de un vector, lo que realmente se está haciendo es declarar un apuntador al inicio del arreglo. Por lo que al enviar un vector como parámetro se enviará la dirección de inicio del vector (en este caso, por las características del lenguaje C, para vectores y matrices sólo se acepta el paso de parámetros por referencia). En el programa 5.3, las funciones `imprime()` y `imprime1()` muestran las formas en las que se puede manejar un vector, como se observa se puede manejar como un vector o como un apuntador, es indistinto.

```
#include <stdio.h>
void imprime(int *);
void imprime1(int []);
main(){
    int i,a[]={1,2,3,4,5};
    for (i=0;i<5;i++)
        printf("%u..",a[i]);
        putchar('\n');
    imprime(a);
    imprime1(a);
    getchar();
}
```

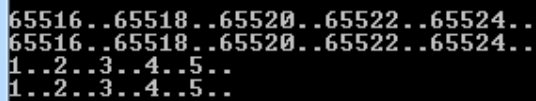
```
void imprime(int b[]){
    int i;
    for (i=0;i<5;i++)
        printf("%u..",b[i]);
    putchar('\n');
```

```

    for (i=0;i<5;i++)
        printf("%d..",*(b+i));
    putchar('\n');
}

void imprime1(int *c){
    int j;
    for(j=0;j<5;j++)
        printf("%d..",c[j]);
}

```



```

65516..65518..65520..65522..65524..
65516..65518..65520..65522..65524..
1..2..3..4..5..
1..2..3..4..5..

```

### Programa 5.3

La primera impresión se invoca en la función -main()- y muestra los cinco espacios de memoria donde se tiene guardado el vector, la segunda impresión se invoca en la función -imprime()-, observe que la dirección de memoria es la misma. Tanto la función -imprime()- como la función imprime1()- muestran una forma indistinta de manejar los vectores (como apuntador o como vector). Observe que el pase de parámetros se realiza sólo con el nombre del vector a comparación de las variables escalares que al simular el paso de parámetros por referencia se tiene que utilizar el prefijo “&” en la función llamadora y el prefijo “\*” en la función llamada,

Un detalle importante, el lenguaje C no verifica los acotamientos tanto inferior como superior, por lo que puede sobre escribir o puede mostrar basura. Observe el programa 5.4:

```

#include <stdio.h>
void imprime(int *);
main(){
    int i ,a[]={1,2,3,4,5};
    imprime(a);
    getchar();
}

void imprime(int b[]){
    int i ;
    for (i=-3;i<8;i++)
        printf("%u..",*(b+i));
    putchar('\n');
    for (i=-3;i<8;i++)
        printf("%d..",*(b+i));
}

```

```

    putchar('\n');
}
65510..65512..65514..65516..65518..65520..65522..65524..65526..65528..65530..
-10..686..-20..1..2..3..4..5..0..344..0..

```

Programa 5.4

En este programa se está declarando e inicializando a un vector con cinco elementos, observe que el comando "for" se ejecuta con una inicialización de -3 hasta 7 (se desborda tanto en la izquierda como en la derecha) mostrando la dirección de memoria y los valores guardados en cada una de esas direcciones:

Al mostrar el contenido del vector, los primeros tres y los últimos tres valores mostrados en pantalla son basura pudiendo suceder conflictos como los comentados en otros capítulos.

### 5.3 Paso de parámetros en matrices estáticas.

Un arreglo multidimensional puede ser visto en varias formas en el lenguaje de programación C, por ejemplo:

Un arreglo de dos dimensiones es un arreglo de una dimensión, donde cada uno de los elementos en sí mismo es un arreglo.

Por lo tanto, la notación:

`A[n][m]`

Nos indica que los elementos del arreglo están guardados renglón por renglón.

Cuando se pasa un arreglo bidimensional a una función se debe especificar el número de columnas ya que el número de renglones es irrelevante.

La razón de lo anterior, es nuevamente los apuntadores, C requiere conocer cuántas son las columnas para que pueda brincar de renglón en renglón en la memoria.

Considerando que una función deba recibir como argumento o parámetro una variable `int a[5][35]` se puede declarar el argumento de la función llamada como:

`f( int a[][35]){ ... }`

o aún como:

`f( int (*a)[35]){ ... }`

En el último ejemplo se requieren los paréntesis `-(*)-` porque el operador `-[]-` tiene precedencia sobre el operador `-*-`.

Por lo tanto:

`int (*a)[35]` declara un apuntador a un arreglo de 35 enteros, y por ejemplo, si hacemos la siguiente referencia `a+2`, nos estamos refiriendo a la dirección del primer elemento que se

encuentra en el tercer renglón de la matriz supuesta, mientras que `int *a[35];` está declarando un arreglo de 35 apuntadores a enteros. Observe el programa 5.5:

```
#include <stdio.h>
#define H      4
void matriz(int[][3]);
void matriz1(int (*)[3]);
main(){
    int a[][3]={1,2,3},{4,5,6},{7,8,9},{10,11,12}};
    matriz (a);
    putchar('\n');
    matriz1(a);
    getchar();
}

void matriz(int (*b)[3]){
    int i,j;
    for (i=0;i<H;i++){
        for (j=0;j<3;j++){
            printf("%d\t",(*b+i)[j]);
            putchar('\n');
        }
    }
}

void matriz1(int b[][3]){
    int i,j;
    for (i=0;i<H;i++){
        for (j=0;j<3;j++){
            printf("%d\t",b[i][j]);

            putchar('\n');
        }
    }
}
```



1	2	3
4	5	6
7	8	9
10	11	12
1	2	3
4	5	6
7	8	9
10	11	12

Programa 5.5

En este programa las funciones `matriz()` y `matriz1()` se emplean indistintamente la notación tipo apuntador como la notación matricial.

Al igual que en vectores, en matrices C no cuida el límite inferior ni el superior. Por ejemplo, el programa 5.6 se muestra la basura que se tiene en memoria cuando la matriz rebasa tanto su límite inferior como su límite superior:

```
#include <stdio.h>
#define H      4
void matriz(int[][3]);
void matriz1(int (*)(3));
main(){
    int a[][3]={1,2,3},{4,5,6},{7,8,9},{10,11,12}};
    matriz(a);
    putchar('\n');
    matriz1(a);
    getchar();
}

void matriz(int (*b)[3]){
    int i,j;
    for (i=-1;i<6;i++){
        for (j=0;j<6;j++){
            printf("%d\t",(*b+i)[j]);
            putchar('\n');
        }
    }
}

void matriz1(int b[][3]){
    int i,j;
    for (i=-1;i<6;i++){
        for (j=-1;j<6;j++){
            printf("%d\t",b[i][j]);
            putchar('\n');
        }
    }
}
```

```

-2      1976998106      16      1      2      3
1       2       3       4       5       6
4       5       6       7       8       9
7       8       9      10      11      12
10      11      12      0       0      2293624
0       0      2293624 4198887 1      5443440
4198887 1      5443440 5444912 -1     2293616

2028240130      -2      1976998106      16      1      2      3
16      1       2       3       4       5       6
3       4       5       6       7       8       9
6       7       8       9      10      11      12
9       10      11      12      0       0      2293624
12      0       0      2293624 4198887 1      5443440
2293624 4198887 1      5443440 5444912 -1     2293616

```

Programa 5.6

#### 5.4 Paso de parámetros con matrices dinámicas.

Si el usuario conoce el tamaño de la matriz hasta el momento de la ejecución, se puede utilizar la memoria en forma dinámica como se analizó en el capítulo 3. El argumento de la matriz se manejará en la función llamada como un apuntador doble, además se debe de incluir el número de hileras y el número de columnas para que la función llamada pueda manejar en forma adecuada la dimensión de la matriz. Estos elementos se muestran en el programa 5.7:

```

#include <stdio.h>
#include <stdlib.h>
void matriz(int **,int,int);
main(){
    int f,c,i,j;
    int **pm;
    printf("Da el numero de hileras=>");
    scanf("%d",&f);
    getchar();
    printf("Da el numero de columnas=>");
    scanf("%d",&c);
    pm=(int **)malloc(sizeof(int *)*f);
    for (j=0;j<c;j++)
        pm[j]=(int*)malloc(sizeof(int)*c);
    for (i=0;i<f;i++)
        for (j=0;j<c;j++)
            pm[i][j]=i*j+1;
    matriz(pm,f,c);
    getchar();
    getchar();
}

```

```

void matriz(int **b, int hil, int col){

```

```

int i,j;
printf("Mostrando la matriz utilizando corchetes\n");
for (i=0;i<hil;i++){
    for (j=0;j<col;j++){
        printf("%d\t",b[i][j]);
        putchar('\n');
    }
    printf("Mostrando la matriz utilizando apuntadores\n");
    for (i=0;i<hil;i++){
        for (j=0;j<col;j++){
            printf("%d\t",*(b+i+j));
            putchar('\n');
        }
    }
}

```

```

Da el numero de hileras=>3
Da el numero de columnas=>4
Mostrando la matriz utilizando corchetes
1      1      1      1
1      2      3      4
1      3      5      7
Mostrando la matriz utilizando apuntadores
1      1      1      1
1      2      3      4
1      3      5      7

```

Programa 5.7

En la función -matriz()- se emplea indistintamente la notación tipo apuntador como la notación matricial.

Un punto interesante de comentar es que, en algunos compiladores el programa compila bien pero no ejecuta en forma apropiada (el programa ejecutó bien en dev-c++, en Turbo C y en Turbo C portable, no siendo así en Borland C).

### 5.5 Paso de parámetros en registros.

Si se desea enviar como parámetro sólo un registro, el paso parámetro será por valor. Por ejemplo, observe el programa 5.8:

```

#include <stdio.h>
#include <string.h>
struct alumno{
    char a[10];
    char b[10];
    int c;
};
void cambia(alumno );
main(){

```

```

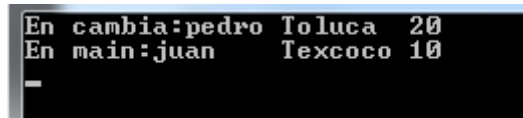
alumno alumnos;
strcpy(alumnos.a,"juan");
strcpy(alumnos.b,"Texcoco");
alumnos.c=10;
cambia(alumnos);
printf("En main:%s\t%s\t%d\n",alumnos.a,alumnos.b,alumnos.c);
getchar();
}

```

```

void cambia(alumno b){
    strcpy(b.a,"pedro");
    strcpy(b.b,"Toluca");
    b.c=20;
    printf("En cambia:%s\t%s\t%d\n",b.a,b.b,b.c);
}

```



```

En cambia:pedro Toluca 20
En main:juan Texcoco 10
_

```

Programa 5.8

Por lo que los cambios realizados dentro de la función -cambia()- sólo se consideran en forma local.

Para poder realizar paso de parámetros por referencia se tiene que realizar el envío de la dirección de memoria en forma explícita con el operador -&- y la recepción del parámetro se realizará con el operador -\*-. Observe el programa 5.9:

```

#include <stdio.h>
#include <string.h>
struct alumno{
    char a[10];
    char b[10];
    int c;
};
void cambia(alumno *);
main(){
    alumno alumnos;
    strcpy(alumnos.a,"juan");
    strcpy(alumnos.b,"Texcoco");
    alumnos.c=10;
    cambia(&alumnos);
    printf("En main:%s\t%s\t%d\n",alumnos.a,alumnos.b,alumnos.c);
}

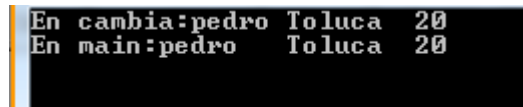
```

```

    getchar();
}

void cambia(alumno *b){
    strcpy(b->a,"pedro");
    strcpy(b->b,"Toluca");
    b->c=20;
    printf("En cambia:%s\t%s\t%d\n",b->a,b->b,b->c);
}

```



```

En cambia:pedro Toluca 20
En main:pedro Toluca 20

```

Programa 5.9

Si se desea enviar como parámetro un arreglo de registros se procede como se muestra en el programa 5.10:

```

#include <stdio.h>
#include <string.h>
struct alumno{
    char a[10];
    char b[10];
    int c;
};

void cambia(alumno *);
void cambia1(alumno[]);
main(){
    int i;
    alumno alumnos[3];
    strcpy(alumnos[0].a,"juan"); strcpy(alumnos[0].b,"Texcoco"); alumnos[0].c=10;
    strcpy(alumnos[1].a,"luis"); strcpy(alumnos[1].b,"Tepexpan"); alumnos[1].c=20;
    strcpy(alumnos[2].a,"lucas"); strcpy(alumnos[2].b,"Tocuila"); alumnos[2].c=10;
    cambia(alumnos);
    putchar('\n');
    for (i=0;i<3;i++)
        printf("En main:%s\t%s\t%d\n",alumnos[i].a,alumnos[i].b,alumnos[i].c);
    putchar('\n');
    cambia1(alumnos);
    putchar('\n');
    for (i=0;i<3;i++)
        printf("En main:%s\t%s\t%d\n",alumnos[i].a,alumnos[i].b,alumnos[i].c);
}

```

```

    getchar();
}

void cambia(alumno b[]){
    int i;
    strcpy(b[0].a,"juana"); strcpy(b[0].b,"Texas"); b[0].c=100;
    strcpy((*b+1).a,"luisa"); strcpy((*b+1).b,"Tampa"); (*b+1).c=200;
    strcpy((b+2)->a,"lola"); strcpy((b+2)->b,"Tulane"); (b+2)->c=300;
    for (i=0;i<3;i++)
        printf("En cambia:%s\t%s\t%d\n",(*b+i).a,(b+i)->b,b[i].c);
}

void cambia1(alumno *c){
    int i;
    strcpy(c[0].a,"Lula"); strcpy(c[0].b,"Taxco"); c[0].c=100;
    strcpy((*c+1).a,"Cuca"); strcpy((*c+1).b,"Sonora"); (*c+1).c=200;
    strcpy((c+2)->a,"Pecche"); strcpy((c+2)->b,"Tula"); (c+2)->c=300;
    for (i=0;i<3;i++)
        printf("En cambia:%s\t%s\t%d\n",(*c+i).a,(c+i)->b,c[i].c);
}

```

```

En cambia:juana Texas 100
En cambia:luisa Tampa 200
En cambia:lola Tulane 300

En main:juana Texas 100
En main:luisa Tampa 200
En main:lola Tulane 300

En cambia:Lula Taxco 100
En cambia:Cuca Sonora 200
En cambia:Pecche Tula 300

En main:Lula Taxco 100
En main:Cuca Sonora 200
En main:Pecche Tula 300

```

Programa 5.10

En este programa, las funciones -cambia()- y -cambia1()- muestran las dos formas validas del paso de parámetro para un arreglo de registros. Observe que se maneja en forma indistinta la notación de apuntadores como la notación de arreglos.

En el siguiente capítulo se mostrará la forma en que se utilizan los registros con manejo de memoria dinámica.

## 5.6 Apuntadores a funciones.

Para hablar de apuntadores a funciones, previamente se establece una dirección a la función.

Cuando se declara una matriz se asume que la dirección es la del primer elemento, del mismo modo, se asume que la dirección de una función será la del segmento de código donde comienza la función. Es decir, la dirección de memoria a la que se transfiere el control cuando se invoca (su punto de comienzo).

Técnicamente un apuntador a función es una variable que guarda la dirección de comienzo de la función. La mejor manera de pensar en el apuntador a función es considerándolo como una especie de “alias” de la función, aunque con una importante cualidad añadida: que **pueden ser utilizados como argumento de otras funciones.**

Por lo que un apuntador a función es un artificio que el lenguaje C utiliza para poder enviar funciones como argumento de una función, la gramática del lenguaje C no permite en principio utilizar funciones en la declaración de parámetros. Un punto importante, ***no está permitido hacer operaciones aritméticas con este tipo de apuntador.***

Lo anterior es útil cuando se deben usar distintas funciones, quizás para realizar tareas similares con los datos. Por ejemplo, se pueden pasar como parámetros los datos y la función que serán usados por alguna función de control. La biblioteca estándar de C tiene funciones para ordenamiento (*qsort()*) y para realizar búsqueda (*bsearch()*), a las cuales se les pueden pasar funciones como parámetros.

La declaración de un apuntador a función se realiza de la siguiente forma:

```
int (* pf)();
```

Donde **-pf-** es un apuntador a una función que no envía parámetros y retorna un tipo de dato entero. Observe que se ha declarado el apuntador y en este momento no se ha dicho a qué variable va a apuntar.

Supongamos que se tiene una función *-int f();-* entonces simplemente se debe de escribir:

```
pf= f;
```

Para que la variable **-pf-** apunte al inicio de la función *-f()-*.

Los apuntadores a funciones se declaran en el área de prototipos, por ejemplo:

```
int f(int);
```

```
int (*fp) (int) =f;
```

En consecuencia, el cuerpo del programa se pueden tener asignaciones como:

```
ans=f(5);
```

```
ans=pf(5);
```

Los cuales son equivalentes.

Observe el programa 5.11:

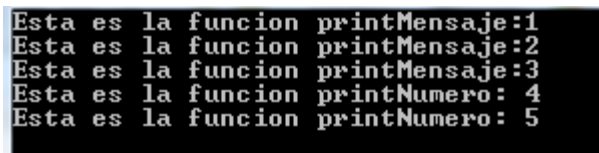
```
#include <stdio.h>
```

```
#include <conio.h>
```

```

void printMensaje(int dato );
void printNumero(int);
void (*funcPuntero)(int);
main(){
    //clrscr();
    printMensaje(1);
    funcPuntero=printMensaje;
    funcPuntero(2);
    funcPuntero(3);
    funcPuntero=printNumero;
    funcPuntero(4);
    printNumero(5);
    getchar();
}
void printMensaje(int dato){
    printf("Esta es la funcion printMensaje:%d\n",dato);
}
void printNumero(int dato){
    printf("Esta es la funcion printNumero: %d\n",dato);
}

```



```

Esta es la funcion printMensaje:1
Esta es la funcion printMensaje:2
Esta es la funcion printMensaje:3
Esta es la funcion printNumero: 4
Esta es la funcion printNumero: 5

```

Programa 5.11

En este programa hemos declarado dos funciones, -printMensaje()- y -printNumero()-, y después -(\*funcPuntero)-, que es un apuntador a una función que recibe un parámetro entero y no devuelve nada (void). Las dos funciones declaradas anteriormente se ajustan precisamente a este perfil, y por tanto pueden ser llamadas por este apuntador. En la función principal, llamamos a la función -printMensaje()- con el valor uno como parámetro, en la línea siguiente asignamos al puntero a función (\*funcPuntero) el valor de -printMensaje()- y utilizamos el apuntador para llamar a la misma función de nuevo. Por tanto, las dos llamadas a la función -printMensaje()- son idénticas gracias a la utilización del puntero -(\*funcPuntero)-.

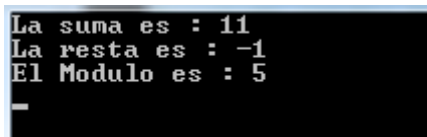
Dado que hemos asignado el nombre de una función a un apuntador a función, y el compilador no da error, el nombre de una función debe ser un apuntador a una función. Esto es exactamente lo que sucede. Un nombre de una función es un apuntador a esa función, pero es un apuntador constante que no puede ser cambiado. Sucede lo mismo con los vectores: el nombre de un vector es un apuntador constante al primer elemento del vector.



El nombre de una función es un apuntador a esa función, podemos asignar el nombre de una función a un apuntador constante, y usar el apuntador para llamar a la función. Pero el valor devuelto, así como el número y tipo de parámetros deben ser idénticos. Muchos compiladores de C y C++ no avisan de las diferencias entre las listas de parámetros cuando se hacen las asignaciones. Esto se debe a que las asignaciones se hacen en tiempo de ejecución, cuando la información de este tipo no está disponible para el sistema.

Veamos el código del programa 5.12:

```
#include <stdio.h>
#include <conio.h>
int MiFuncionSuma(int,int);
int MiFuncionResta(int,int);
int MiFuncionModulo(int,int);
main(){
    int (*ptrFn) (int,int);
    ptrFn = MiFuncionSuma;
    printf("La suma es : %d\n", ptrFn(5, 6));
    ptrFn = MiFuncionResta;
    printf("La resta es : %d\n", ptrFn(5, 6));
    ptrFn = MiFuncionModulo;
    printf("El Modulo es : %d\n", ptrFn(5, 6));
    getch();
}
int MiFuncionSuma(int a,int b){
    return a + b;
}
int MiFuncionResta(int a,int b){
    return a - b;
}
int MiFuncionModulo(int a,int b){
    return a % b;
}
```



```
La suma es : 11
La resta es : -1
El Modulo es : 5
_
```

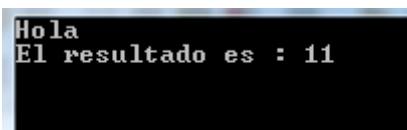
Programa 5.12

En el programa 5.13 se observa la declaración del apuntador dentro de la función –main()– y se muestra la forma en que se puede enviar como parámetro un apuntador a función que retorna a su vez un apuntador tipo entero.

```
#include <stdio.h>
#include <stdlib.h>
int* MiFuncionSuma(int,int);
void Imprime_Resultado(int *);
main(){
    printf("Hola\n");
    int* (*ptrFnSum) (int,int);
    void (*ptrFnRes) (int*);
    ptrFnSum = MiFuncionSuma;
    ptrFnRes = Imprime_Resultado;
    ptrFnRes(ptrFnSum(5, 6));
    getchar();
}

int* MiFuncionSuma(int a,int b) {
    int *calculo;
    calculo =(int *)malloc(sizeof(int));
    *calculo = a + b;
    return calculo;
}

void Imprime_Resultado(int *a){
    printf("El resultado es : %d \n",*a);
}
```



```
Hola
El resultado es : 11
```

Programa 5.13

Como hemos visto en los programas anteriores, lo que hacemos es: primero, definir nuestras funciones, con o sin parámetros, luego definimos las variables del tipo apuntador a función, por último enlazamos nuestras variables a la dirección de memoria del código de las funciones antes que las invoquemos. El enlace lo debemos hacer a la función, en consecuencia podemos enlazarlo sólo colocando el nombre de la función o usando el operador de dirección –&– seguido del nombre de la función. Por último, trabajamos únicamente con la función definida.

Ahora se realizará un ejemplo donde la función tiene como parámetro a un apuntador a una función.

Primero se definirá la variable tipo apuntador a función:

```
int (*ptrFn) (int, int);
```

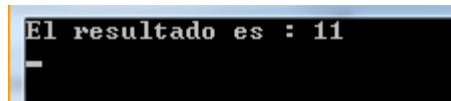
Ahora se definirá una variable tipo apuntador a función que tiene tres parámetros, el primer parámetro es el apuntador tipo función:

```
void (*ptrFnD)(int (*ptrFn)(int, int), int, int);
```

En el programa 5.14 se mostrará el empleo de este tipo de declaración:

```
#include <stdio.h>
int MiFuncionSuma(int, int);
void Imprime_Resultado(int (*)(int, int), int, int);

main(){
void (*ptrFnRes) (int (*)(int,int), int x, int y);
ptrFnRes = &Imprime_Resultado;
ptrFnRes(MiFuncionSuma, 5, 6);
getchar();
}
int MiFuncionSuma(int a,int b){
    return a+b;
}
void Imprime_Resultado(int (*funcion)(int , int),int x,int y){
    printf("El resultado es : %d \n",funcion(x,y));
}
```



Programa 5.14

Considere detenidamente las declaraciones de los siguientes ejemplos (en todos ellos fptr es un apuntador a función de tipo distinto de los demás).

```
void (*fptr)();
```

fptr es un puntero a una función, sin parámetros, que devuelve **void**.

```
void (*fptr)(int);
```

fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.

```
int (*fptr)(int, char);
```

fptr es puntero a función, que acepta un **int** y un **char** como argumentos y devuelve un **int**.

```
int* (*fptr)(int*, char*);
```

fptr es puntero a función, que acepta sendos punteros a **int** y **char** como argumentos, y devuelve un puntero a **int**.

Cuando el valor devuelto por la función es a su vez un puntero (a función, o de cualquier otro tipo), la notación se complica un poco más:

```
int const * (*fptr)();
```

fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un **int** constante

```
float *(*fptr)(char))(int);
```

fptr es un puntero a función que recibe un **char** como argumento y devuelve un puntero a función que recibe un **int** como argumento y devuelve un **float**.

```
void * (*(*fptr)(int))[5];
```

fptr es un puntero a función que recibe un **int** como argumento y devuelve un puntero a un array de 5 punteros-a-**void** (genéricos).

```
char *(*fptr)(int, float))();
```

fptr es un puntero a función que recibe dos argumentos (**int** y **float**), devolviendo un puntero a función que no recibe argumentos y devuelve un **char**.

```
long *(*(*fptr())[5])();
```

fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un array de 5 punteros a función que no reciben ningún parámetro y devuelven **long**.

```
void (*fptr)();
```

fptr es un puntero a una función, sin parámetros, que devuelve **void**.

```
void (*fptr)(int);
```

fptr es un puntero a función que recibe un **int** como parámetro y devuelve **void**.

```
int (*fptr)(int, char);
```

fptr es puntero a función, que acepta un **int** y un **char** como argumentos y devuelve un **int**.

```
int* (*fptr)(int*, char*);
```

fptr es puntero a función, que acepta sendos punteros a **int** y **char** como argumentos, y devuelve un puntero a **int**.

```
int (*afptr[10])(int);
```

afptr es una matriz de 10 apuntadores a función

Ejercicios:

1. Comente el paso de parámetros por valor y el paso de parámetros por referencia.
2. En el lenguaje de programación C, comente el pase de parámetros por referencia para variables escalares, vectoriales y registros.
3. Comente la diferencia entre variables locales, variables globales y variables estáticas.
4. Escribir un programa donde se realice tanto pase de parámetros por valor como por referencia.
5. Para un vector marque la diferencia o similitud entre los siguientes códigos:
  - a. `void f(int b[]){ ... }`
  - b. `void f1(int *b){ ... }`
6. Indique la diferencia o similitud entre los siguientes códigos:
  - a. `void f(int a[][3]){ ... }`
  - b. `void f(int (*a)[3]){ ... }`
7. Explique y diagrame la forma en que el lenguaje de programación C maneja una matriz de N X M.
8. Comente los parámetros del siguiente prototipo:  
`Void m(int ***, int);`
9. Comente la diferencia o similitud entre el pase de parámetros cuando es un registro o una matriz.
10. Comente los siguientes prototipos:
  - a. `Void (*fp)(int)`
  - b. `Void ir(int(*) (int, int), int);`
11. Comente las siguientes declaraciones:
  - a. `Int *(*fpt)(int *, char *);`
  - b. `Float (*(fpt)(char))(int);`
  - c. `Void *(*fpt)(int)[5];`

## SEXTO CAPITULO: LISTAS ENLAZADAS.

### Resumen.

Este capítulo explica completamente la segunda unidad de competencias. Fundamentalmente se introduce las formas más simples de realizar un **enlace** de registros en forma dinámica. Este capítulo es fundamental ya que en muchas aplicaciones, como en los compiladores, por poner un ejemplo, se requiere comprender este tipo de enlaces.

Las ideas básicas se tomaron de:

(Cairó/Gardati, 2000)

(Deitel & Deitel, 1995)

(Ceballos, 2015)

El pseudocódigo para realizar la programación de las estructuras mostradas en este capítulo se tomó fundamentalmente de (Cairó/Gardati, 2000).

### 6.1. Programación de listas enlazadas

Una lista es una colección de elementos llamados generalmente nodos. El orden entre los nodos se establece por medio de apuntadores (realizando el enlace entre nodos), es decir, direcciones o referencias a otros nodos. Las operaciones más comunes son:

- Recorrido de la lista.
- Inserción de un elemento.
- Borrado de un elemento.
- Búsqueda de un elemento.

Los tipos de datos abstractos más conocidos son la pila, la lista ordenada, la lista doblemente enlazada ordenada y el árbol binario. Y son un ejemplo claro del manejo de memoria en forma dinámica. Por lo que en las siguientes sub-secciones se mostrará la programación de estos elementos.

### 6.2 Programación de una pila

Una pila (stack en inglés) es una lista de elementos a la cual se pueden insertar o eliminar elementos sólo por uno de los extremos (apilar o retirar un elemento de la pila). En consecuencia, los elementos de una pila serán eliminados en orden inverso al que se insertaron. Es decir, el último elemento que se mete en la pila es el primero que se saca.

Debido al orden en el cual se insertan o eliminan elementos de una pila a esta estructura también se le conoce como estructura LIFO (Last-In, First-Out: Último en entrar, primero en salir).

Las pilas pertenecen al grupo de estructuras de datos lineales, porque los componentes ocupan lugares sucesivos en la estructura.

Algunos de los ejemplos donde se utiliza la pila son:

- Evaluación de expresiones en notación posfija
- Reconocedores sintácticos de lenguajes independientes del contexto

- Implementación de la recursividad.

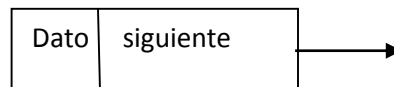
Las operaciones elementales que pueden realizarse en la pila son:

- Colocar un elemento en la pila
- Eliminar un elemento de la pila
- Mostrar la pila

La estructura a usar en el programa es la siguiente:

```
struct nodo{
    int dato;
    nodo * siguiente;
}
```

Esta estructura dinámica se puede representar en forma gráfica de la siguiente manera:

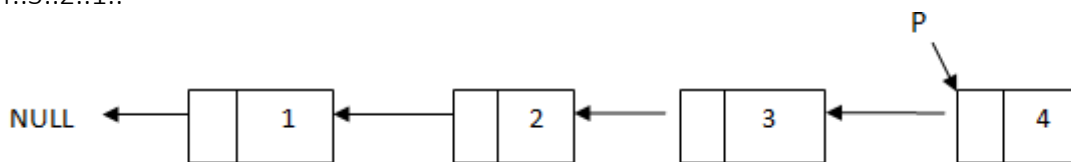


Observe que la variable **–siguiente–** es un apuntador a la misma estructura, esto nos permite poder realizar un enlace con estructuras del mismo tipo (de ahí el nombre de estructuras enlazadas).

En la figura 6.1 se muestra la forma en que se manipula la pila, por lo que el primer elemento introducido es la estructura donde la variable **–dato–** guarda el valor uno y el último elemento en la pila es la estructura donde la variable **–dato–** guarda el valor cuatro.

La impresión de la pila será:

4..3..2..1..



Al eliminar un elemento de la pila, ésta se verá de la siguiente forma:

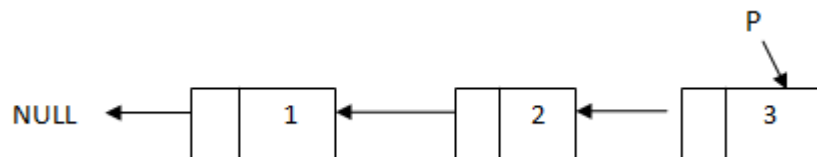


Figura 6.1

El programa 6.1 muestra el código en C de una pila en forma dinámica en el que se permite insertar, mostrar o eliminar elementos:

```

#include<stdio.h>
#include<stdlib.h>
struct nodo{
    int dato;
    nodo * siguiente;
};

//PROTOTIPOS
int menu();
void * crear (void *);
void * eliminar(void *);
void mostrar(void *);

main(){
    void *p=NULL;
    int eleccion;
    do{
        eleccion=menu ();
        switch(eleccion){
            case 1:p=crear(p);continue;
            case 2:p=eliminar(p);break;
            case 3:mostrar(p);continue;
            default:printf("FIN DE LAS OPERACIONES");
        }
    }while(eleccion<4);
    return 0;
}

int menu(){
    int eleccion;
    do{
        printf("\n\t\tMENU PRINCIPAL\n");
        printf("\t1.-Introducir un elemento a la pila\n");
        printf("\t2.-Eliminar un elemento de la pila\n");
        printf("\t3.-Mostrar el contenido de la pila\n");
        printf("\t4.-Salir\n");
        scanf("%d",&eleccion);
    }while(eleccion<1 || eleccion>4);
    putchar('\n');
    return(eleccion);
}

void *crear(void *p){
    nodo *q;
    putchar('\n');

```



```

printf("Indica el valor a introducir a la pila=>");
q=(nodo*)malloc(sizeof(nodo));
scanf("%d",&q->dato);
q->siguiente=NULL;
if(p==NULL) //La pila está vacía
    p=q;
else{
    q->siguiente=(nodo*)p;
    p=q;
}
return(p);
}

```

```

void * eliminar(void * s){
nodo *p;
if(s==NULL)
    printf("\npila vacia\n");
else{
    p=(nodo*)s;
    s=p->siguiente;
    free(p);
}
return(s);
}

```

```

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
    printf("PILA VACIA");
else
    do{
        printf("%d..",s->dato);
        s=s->siguiente;
    }while(s!=NULL);
}

```

```

MENU PRINCIPAL
1.-Introducir un elemento a la pila
2.-Eliminar un elemento de la pila
3.-Mostrar el contenido de la pila
4.-Salir
1
Indica el valor a introducir a la pila=>1
MENU PRINCIPAL
1.-Introducir un elemento a la pila
2.-Eliminar un elemento de la pila
3.-Mostrar el contenido de la pila
4.-Salir
1
Indica el valor a introducir a la pila=>2
MENU PRINCIPAL
1.-Introducir un elemento a la pila
2.-Eliminar un elemento de la pila
3.-Mostrar el contenido de la pila
4.-Salir
1
Indica el valor a introducir a la pila=>3
MENU PRINCIPAL
1.-Introducir un elemento a la pila
2.-Eliminar un elemento de la pila
3.-Mostrar el contenido de la pila
4.-Salir
3
3..2..1..
MENU PRINCIPAL
1.-Introducir un elemento a la pila
2.-Eliminar un elemento de la pila
3.-Mostrar el contenido de la pila
4.-Salir

```

Programa 6.1

Cada vez que se ejecuta la función –crear()-, en la instrucción:

```
q=(nodo*)malloc(sizeof(nodo));
```

la función –sizeof()- indica el tamaño de la estructura –nodo- en bytes. Posteriormente, la función –malloc()- reserva memoria del tamaño indicado en la función –sizeof()- y finalmente la memoria reservada se utilizará como tipo nodo y se regresa un apuntador a una variable del tipo apuntador a nodo.

Para poder mantener la pila en memoria se requiere un apuntador principal al final de la pila, el apuntador utilizado en el programa es del tipo –void-, por lo que se requiere que en cada función invocada realizar un casting para la manipulación de la pila. En la función –crear()- se tiene el casting:

```
q->siguiente=(nodo*)p;
```

En esta instrucción, además del casting, se realiza el enlace entre dos estructuras en forma dinámica. Observe que la variable –p- es un apuntador del tipo void y la variable –q- es un apuntador del tipo nodo, si se realiza una igualdad de la forma:

```
q->siguiente=p;
```

se tendrá un error del tipo mismatch (las variables no son compatibles), esto se entiende ya que se está asignado una variable del tipo general (void) a una variable del tipo específico (nodo \*). Pero:

p=q;

No genera error del tipo mismatch ya que se está asignando la dirección de memoria de un apuntador tipo nodo a un apuntador sin tipo. (Recordará que un apuntador sin tipo puede guardar la dirección de memoria de cualquier variable, pero no puede ir a esa dirección de memoria)

### 6.3 Traducción de una expresión infija a postfija.

Un ejemplo clásico de una pila es el transformar una expresión algebraica infija a postfija.

Para comprender que es una expresión infija o postfija se revisarán algunos conceptos:

- Dada la expresión  $A + B$  se dice que está en notación infija, y su nombre se debe a que el operador (+) se localiza entre los dos operandos (A, B)
- Dada la expresión  $AB+$  se dice que está en notación postfija, ya que el operador (+) se localiza después de los operandos (A, B).
- Dada la expresión  $+AB$  se dice que está en notación prefija ya que el operador (+) se localiza antes de los operandos (A, B).

La ventaja de usar expresiones en notación polaca (o postfija) o prefija radica en que no son necesarios los paréntesis para indicar el orden de los operadores, ya que éste queda establecido por la ubicación del operador con respecto al operando.

Para convertir una expresión dada de notación infija a postfija (o prefija), deberán de establecerse ciertas condiciones:

- Solamente se manejarán los siguientes operadores (están dados ordenadamente de mayor a menor según su prioridad de ejecución):
  - ^ (Potencia)
  - \* / % (multiplicación, división y módulo)
  - + - (suma y resta)
- Los operadores de más alta prioridad se ejecutan primero.
- Si hubiera en una expresión dos o más operadores de igual prioridad, entonces se procederá a evaluar de izquierda a derecha. (Para cumplir con el punto anterior y el actual, se utilizará una pila para manejar las prioridades en forma adecuada.)
- Las sub expresiones que se localicen dentro de paréntesis tienen más prioridad que cualquier otra operación

En la tabla 6.1 se presenta, paso a paso, un ejemplo de conversión de expresión infija a postfija utilizando una pila como apoyo.

Tabla 6.1

PASO	EXPR. INFIJA	SIMBOLO ANALIZADO	PILA	EXPRESIÓN POSTFIJA
0	$(X+Z)*W/T^AY-V$			
1	$X+Y)*W/T^AY-V$	(	(	
2	$+Y)*W/T^AY-V$	X	(	X
3	$Y)*W/T^AY-V$	+	(+	X
4	$) *W/T^AY-V$	Z	(+	XZ
5	$*W/T^AY-V$	)		XZ+
6	$W/T^AY-V$	*	*	XZ+
7	$/T^AY-V$	W	*	XZ+W
8	$T^AY-V$	/	/	XZ+W*
9	$^AY-V$	T	/	XZ+W*T
10	$Y-V$	^	/^	XZ+W*T
11	$-V$	Y	/^	XZ+W*TY
12	$V$	-	-	XZ+W*TY^/
13		V	-	XZ+W*TY^/V-

El programa 6.2 muestra el algoritmo para transformar una expresión infija a una expresión postfija. Se introdujo una función llamada “topepila” que permite ver lo que se tiene en tope de pila sin realizar algún movimiento en ella. También se introdujo una función que evalúa la prioridad del operador que se está analizando actualmente con respecto al operador que se localiza en tope de pila. Las funciones “crear()” y “eliminar()” son muy parecidas a las funciones del mismo nombre al programa 6.1 que maneja una pila. La función “leer()” lee un arreglo de caracteres en forma dinámica.

```
//El apuntador principal apunta al último elemento en pila
#include<stdio.h>
#include<stdlib.h>
struct nodo{
    char dato;
    nodo * siguiente;
};

//PROTOTIPOS
int prioridad(char);           //Indica la prioridad de los operadores
char* leer(char[],int);       //Lee un arreglo de caracteres en forma dinámica
void * crear (void *, char);  //introduce un elemento a pila
void * eliminar(void *);      //Elimina un elemento del tope de pila
void imprime(char[]);         //Imprime la expresión infija
void postfija(char[]);        //El algoritmo de transformación de infija a postfija
char topepila(void*);         //Retorna lo que se tiene en tope de pila
```

```

main(){
char * a;
int i=0;
printf("Da la expresion infija:\n");
do
    a=leer(a,i++);
while(a[i-1]!='\n');
imprime(a);
postfija(a);
return 0;
}

char * leer(char a[],int i){
    char *b;
    b=(char *)malloc(sizeof(char)*(i+1));
    b[i]=getchar();
    for (int j=0;j<i;j++)
        b[j]=a[j];
    free (a);
    return b;
}

void imprime(char x[]){
    int i=0;
    putchar('\n');
    printf("La expresión infija es:\n");
    do
        putchar(x[i++]);
    while(x[i-1]!='\n');
}

void postfija(char a[]){
    void *p=NULL;
    char c;
    int x=0;
    printf("La expresión postfija es:\n");
    while (a[x]!='\n'){
        if (a[x]=='(')
            p=crear(p,'(');
        else if(a[x]==')'){
            while((c=topepila(p))!='('){
                putchar(c);
                p=eliminar(p);
            }
        }
        putchar(a[x]);
        x++;
    }
}

```

```

        }
        p=eliminar(p);
    }
    else if(a[x]>='a' && a[x]<='z')
        putchar(a[x]);
    else {
        while (topepila(p)!='&' && prioridad(a[x])<=prioridad(topepila(p))){
            putchar(topepila(p));
            p=eliminar(p);
        }
        p=crear(p,a[x]);
    }
    x++;
}
while ((c=topepila(p))!='&'){
    putchar(c);
    p=eliminar(p);
}
}

```

```

int prioridad(char c){
    int x;
    switch(c){
        case '^':x=3; break;
        case '*':
        case '/':
        case '%':x=2; break;
        case '+':
        case '-':x=1; break;
        case ' ':x=0; break;
    }
    return x;
}

```

```

void *crear(void *p, char a){
    nodo *q;
    q=(nodo*)malloc(sizeof(nodo));
    q->siguiente=NULL;
    q->dato=a;
    if(p==NULL) //La pila está vacía
        p=q;
    else{
        q->siguiente=(nodo*)p;
    }
}

```

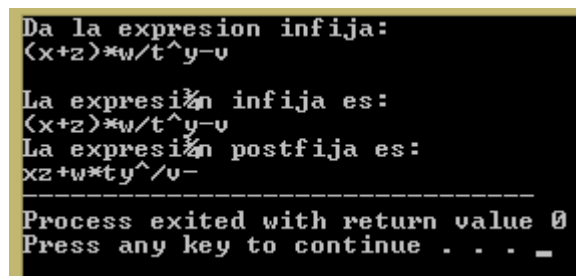
```

    p=q;
}
return(p);
}

char topepila(void *p){
    nodo *q=(nodo*)p;
    if (p==NULL)
        return '&';
    else
        return q->dato;
}

void * eliminar(void * s){
    nodo *p;
    if(s==NULL)
        printf("\npila vacia\n");
    else{
        p=(nodo*)s;
        s=p->siguiente;
        free(p);
    }
    return(s);
}

```



```

Da la expresion infija:
(x+z)*w/t^y-u

La expresion infija es:
(x+z)*w/t^y-u
La expresion postfija es:
xz+w*ty^/u-

Process exited with return value 0
Press any key to continue . . . _

```

Programa 6.2

#### 6.4 Programación de una lista simplemente enlazada ordenada.

Una lista ordenada es una lista enlazada con un orden, siendo el orden una estructura interna que debe mantenerse en todo momento.

Las operaciones elementales que pueden realizarse en la lista ordenada son:

- Colocar un elemento en la lista
- Eliminar un elemento de la lista
- Mostrar la lista

Para colocar un elemento en la lista ordenada se deben de observar cuatro posibles casos:

- La lista está vacía.
- El elemento a colocar es menor a todos los elementos que se encuentran en la lista
- El elemento a colocar es mayor a todos los elementos que se encuentran en la lista
- El elemento a colocar debe de estar entre la lista.

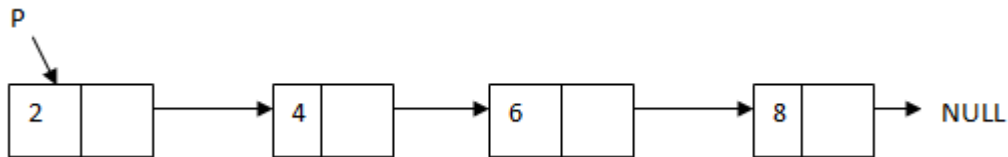
Cada caso se observa en la función –crear()- del siguiente programa.

Para eliminar un elemento de la lista se tienen los siguientes casos:

- El elemento a eliminar es el primero y único de la lista
- El elemento a eliminar es el primero y no es el único
- El elemento a eliminar es el último de la lista
- El elemento a eliminar se encuentra entre la lista.

Cada caso se observa en la función –eliminar()- del siguiente programa.

Por lo que el programa 6.3 ordena los elementos de menor a mayor creando el espacio del elemento a guardar en forma dinámica. Además, permite eliminar un elemento sin perder el orden establecido:



```
#include<stdio.h>
#include<stdlib.h>
struct nodo{
    int dato;
    nodo * siguiente;
};
//PROTOTIPOS
int menu();
void * crear (void * );
void * eliminar(void * );
void mostrar(void * );

main(){
    void *p=NULL;
    int eleccion;
    do{
        eleccion=menu ();
        switch(eleccion){
            case 1:p=crear(p);continue;
            case 2:p=eliminar(p);break;
```



```

        case 3:mostrar(p);continue;
        default:printf("FIN DE LAS OPERACIONES");
    }
}while(eleccion<4);
return 0;
}

int menu(){
int eleccion;
do{
    printf("\n\t\tMENU PRINCIPAL\n");
    printf("\t1.-Introducir un elemento a la lista\n");
    printf("\t2.-Eliminar un elemento de la lista\n");
    printf("\t3.-Mostrar el contenido de la lista\n");
    printf("\t4.-Salir\n");
    scanf("%d",&eleccion);
}while(eleccion<1 || eleccion>4);
putchar('\n');
return(eleccion);
}

void *crear(void *p){
nodo *q,*aux,*aux1;
int x;
putchar('\n');
printf("Indica el valor a introducir a la lista=>");
scanf("%d",&x);
q=(nodo*)malloc(sizeof(nodo));
q->dato=x;
q->siguiente=NULL;
if(p==NULL)        //Lista vacía
    p=q;
else{
    aux=(nodo*)p;
    if (x<aux->dato){        //Elemento menor a cualquier elemento de la lista
        q->siguiente=aux;
        p=q;
    }
    else{        //el elemento se colocará entre la lista o será el último
        while(aux!=NULL&&aux->dato<x){
            aux1=aux;
            aux=aux->siguiente;
        }
        aux1->siguiente=q;
        q->siguiente=aux;
    }
}
}

```

```

    }
}
return(p);
}

void * eliminar(void * s){
if(s==NULL)
    printf("\nLISTA VACIA\n");
else{
    nodo *p,* aux;
    int x;
    printf("Da el elemento a eliminar=>");
    scanf("%d",&x);
    aux=p=(nodo*)s;
    if(p->siguiente==NULL&&aux->dato==x)//solo hay un elemento en la lista
        s=NULL;
    else {
        while(p->siguiente!=NULL && p->dato<x){
            aux=p;
            p=p->siguiente;
        }
        if (p!=NULL)
            if (p->dato==x && p==s) //No existió movimiento
                s=p->siguiente;
            else if(p->dato==x)
                aux->siguiente=p->siguiente;
            else
                printf("DATO NO LOCALIZADO\n");
        else
            printf("DATO NO ENCOTRADO\n");
    }
    free(p);
}
return(s);
}

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
    printf("LISTA VACIA");
else
    do{

```

```

        printf("%d..",s->dato);
        s=s->siguiente;
    }while(s!=NULL);
}

```

```

          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista
4.-Salir
1
Indica el valor a introducir a la lista=>3
          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista
4.-Salir
1
Indica el valor a introducir a la lista=>1
          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista
4.-Salir
1
Indica el valor a introducir a la lista=>2
          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista
4.-Salir
3
1..2..3..

```

Programa 6.3

## 6.5 Programación de una lista doblemente enlazada ordenada

La lista puede ser doblemente enlazada por lo que permite ordenar la lista y recorrerla de izquierda a derecha y de derecha a izquierda, permitiendo en este caso mostrar los números ingresados en forma ascendente o en forma descendente. (Es el preámbulo para poder programar un árbol binario)

Los casos para la inserción y eliminación son similares a las listas simplemente enlazadas ordenadas con la única diferencia de que se tiene que realizar el doble enlace al momento de introducir un nuevo elemento.

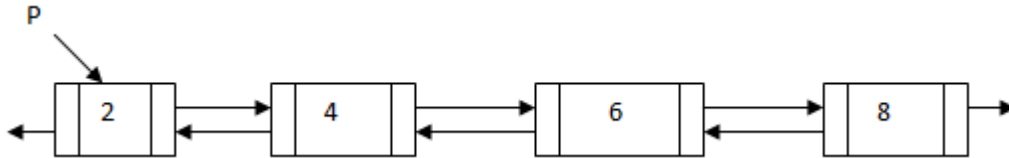
Por lo que la estructura tiene dos apuntadores:

```

struct nodo{
    int dato;
    nodo * izq; //Apuntador al nodo izquierdo
    nodo * der; //Apuntador al nodo derecho
}

```

Gráficamente quedaría de la siguiente forma:



El programa 7.4 permite introducir elementos en forma ordenada, mostrar elementos en forma ascendente y en forma descendente y eliminación de elementos.

```

#include<stdio.h>
#include<stdlib.h>
struct nodo{
    int dato;
    nodo * izq;
    nodo * der;
};

//PROTOTIPOS
int menu();
void * crear (void * );
void * eliminar(void * );
void mostrarasc(void * );
void mostrardesc(void * );
main(){
    void *p=NULL;
    int eleccion;
    do{
        eleccion=menu();
        switch(eleccion){
            case 1:p=crear(p);continue;
            case 2:p=eliminar(p);break;
            case 3:mostrarasc(p);continue;
            case 4:mostrardesc(p);continue;
            default:printf("FIN DE LAS OPERACIONES");
        }
    }while(eleccion<5);
    return 0;
}

int menu(){
    int eleccion;
    do{
        printf("\n\t\tMENU PRINCIPAL\n");
        printf("\t1.-Introducir un elemento a la lista\n");
        printf("\t2.-Eliminar un elemento de la lista\n");
        printf("\t3.-Mostrar el contenido de la lista en forma ascendente\n");
    }

```

```

        printf("\t4.-Mostrar el contenido de la lista en forma descendente\n");
        printf("\t5.-Salir\n");
        scanf("%d",&eleccion);
    }while(eleccion<1 || eleccion>5);
    putchar('\n');
    return(eleccion);
}

void *crear(void *p){
    nodo *q,*aux,*aux1;
    int x;
    putchar('\n');
    printf("Indica el valor a introducir a la lista=>");
    scanf("%d",&x);
    q=(nodo*)malloc(sizeof(nodo));
    q->dato=x;
    q->izq=NULL;
    q->der=NULL;
    if(p==NULL)          //La lista está vacía
        p=q;
    else{
        aux=(nodo*)p;
        if (x<aux->dato){      //El dato es menor a todos los datos localizados en la lista
            q->der=aux;
            aux->izq=q;
            p=q;
        }
        else{
            while(aux!=NULL && aux->dato<x){
                aux1=aux;
                aux=aux->der;
            }
            aux1->der=q;
            q->izq=aux1;
            if (aux!=NULL){ //El dato se colocará entre la lista
                q->der=aux;
                aux->izq=q;
            }
        }
    }
    return(p);
}

void * eliminar(void * s){
    if(s==NULL)
        printf("\nLISTA VACIA\n");

```

```

else{
    nodo *p,* aux,*aux1;
    int x;
    printf("Da el elemento a eliminar=>");
    scanf("%d",&x);
    p=(nodo*)s;
    if (p->der==NULL && p->dato==x)    //Solo hay un elemento en la lista
        s=NULL;
    else {
        while(p->der!=NULL && p->dato<x){
            aux=p;
            p=p->der;
        }
        if (p!=NULL){
            if (p->dato==x && p==s){ //No existió movimiento
                s=aux1=p->der;
                if(aux1!=NULL)
                    aux1->izq=NULL;
            }
            else if (p->dato==x){
                aux1=aux->der=p->der;
                if (aux1!=NULL)
                    aux1->izq=aux;
            }
            else
                printf("DATO NO LOCALIZADO\n");
        }
        else
            printf("DATO NO ENCOTRADO\n");
    }
    free(p);
}
return(s);
}
void mostrarasc(void *p){
    nodo *s;
    s=(nodo *)p;
    if (p==NULL)
        printf("LISTA VACIA");
    else
        do{
            printf("%d..",s->dato);
            s=s->der;
        }while(s!=NULL);
}

```

```

}
void mostrardesc(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
    printf("LISTA VACIA");
else{
    while(s->der!=NULL)
        s=s->der;
    do{
        printf("%d..",s->dato);
        s=s->izq;
    }while(s!=NULL);
}
}
}

```

```

          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista en forma ascendente
4.-Mostrar el contenido de la lista en forma descendente
5.-Salir
1
Indica el valor a introducir a la lista=>3

          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista en forma ascendente
4.-Mostrar el contenido de la lista en forma descendente
5.-Salir
1
Indica el valor a introducir a la lista=>1

          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista en forma ascendente
4.-Mostrar el contenido de la lista en forma descendente
5.-Salir
1
Indica el valor a introducir a la lista=>2

          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista en forma ascendente
4.-Mostrar el contenido de la lista en forma descendente
5.-Salir
3
1..2..3..

          MENU PRINCIPAL
1.-Introducir un elemento a la lista
2.-Eliminar un elemento de la lista
3.-Mostrar el contenido de la lista en forma ascendente
4.-Mostrar el contenido de la lista en forma descendente
5.-Salir
4
3..2..1..

```

Programa 6.4

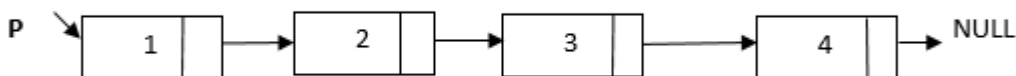
## 6.6 Programación de una fila o cola.

Una fila es una lista de elementos en la que éstos se introducen por un extremo y se eliminan por el otro extremo. Los elementos se eliminan en el mismo orden en el que se insertaron. Por lo tanto, el primer elemento que entra a la cola será el primero en salir. Debido a esta característica, las colas también reciben el nombre de estructuras FIFO (First-In, First-Out).

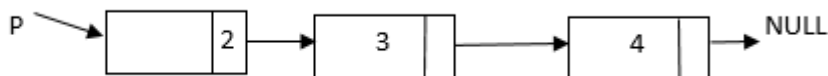
Existen numerosos casos de colas en la vida real: Las personas esperando en la fila de tortillas, o en la fila de un supermercado, la fila de autos esperando el cambio de color de un semáforo, etc. Al igual que las pilas, las colas no existen como estructuras de datos estándares en los lenguajes de programación. Las colas se pueden representar mediante el uso de:

- Arreglos.
- Listas enlazadas.

En este caso, se utilizarán listas enlazadas para la programación de tal estructura. En la siguiente figura se muestra la forma en que se utilizará el apuntador de referencia para poder manejar la cola:



Al eliminar un elemento de la fila, ésta se verá de la siguiente forma:



El programa 6.5 muestra el código en C de una cola en forma dinámica en el que se permite insertar, mostrar o eliminar elementos:

```
//El apuntador principal apunta al primer elemento de la fila
#include<stdio.h>
#include<stdlib.h>
struct nodo{
    int dato;
    nodo * siguiente;
```



```

};

//PROTOTIPOS
int menu();
void * crear (void *);
void * eliminar(void *);
void mostrar(void *);

main(){
void *p=NULL;
int eleccion;
do{
    eleccion=menu();
    switch(eleccion){
        case 1:p=crear(p);continue;
        case 2:p=eliminar(p);break;
        case 3:mostrar(p);continue;
        default:printf("FIN DE LAS OPERACIONES");
    }
}while(eleccion<4);
return 0;
}

int menu(){
int eleccion;
do{
    printf("\n\t\tMENU PRINCIPAL\n");
    printf("\t1.-Introducir un elemento a la fila\n");
    printf("\t2.-Eliminar un elemento de la fila\n");
    printf("\t3.-Mostrar el contenido de la fila\n");
    printf("\t4.-Salir\n");
    scanf("%d",&eleccion);
}while(eleccion<1 || eleccion>4);
putchar('\n');
return(eleccion);
}

void *crear(void *p){
nodo *q,*aux;
putchar('\n');
printf("Indica el valor a introducir a la fila=>");
q=(nodo*)malloc(sizeof(nodo));
scanf("%d",&q->dato);
q->siguiente=NULL;

```

```

if(p==NULL) //La fila está vacía
    p=q;
else{
    aux=(nodo*)p;
    while(aux->siguiente!=NULL)
        aux=aux->siguiente;
    aux->siguiente=q;
}
return(p);
}

```

```

void * eliminar(void * s){
nodo *p,* aux;
if(s==NULL)
    printf("\nfila vacia\n");
else{
    p=(nodo*)s;
    s=p->siguiente;
    free(p);
}
return(s);
}

```

```

void mostrar(void *p){
nodo *s;
s=(nodo *)p;
if (p==NULL)
    printf("FILA VACIA");
else
    do{
        printf("%d..",s->dato);
        s=s->siguiente;
    }while(s!=NULL);
}

```

### Programa 6.5

#### 6.7 Aplicaciones de listas enlazadas.

##### 6.7.1 Aplicaciones en pilas

El concepto de pila está ligado a la computación. Una aplicación de las pilas puede verse en Teoría de Computación y Compiladores, principalmente en la programación de los autómatas tipo pila y el apoyo para poder programar un árbol sintáctico (Parser Tree). En los lenguajes ensambladores y en los microcontroladores donde por lo general se tienen la

instrucción Push y Pop para introducir y obtener datos de una pila. Por ejemplo, si el registro AX se carga previamente con el valor 5, una instrucción Push AX almacenaría el valor en la última posición de la pila. Por otro lado la instrucción Pop saca el último dato almacenado en la pila y lo coloca en el operando. Siguiendo el ejemplo anterior, la instrucción Pop BX obtendrá el número cinco y lo almacenará en el registro BX.

Las funciones recursivas (capítulo 7) tienen el inconveniente de ser menos eficiente que una función iterativa. Si la eficiencia es un parámetro crítico, y la función se va a ejecutar frecuentemente, conviene escribir una solución iterativa. La recursión se puede simular con el uso de pilas para transformar un programa recursivo en iterativo. Las pilas se usan para almacenar los valores de los parámetros del subprograma, los valores de las variables locales, y los resultados de la función.

#### 6.7.2 Aplicaciones de colas.

El concepto de cola está ligado a computación. Una aplicación de las colas puede verse en las colas de impresión. Cuando hay una sola impresora para atender a varios usuarios, puede suceder que algunos de ellos soliciten los servicios de impresión al mismo tiempo o mientras el dispositivo está ocupado. En estos casos se forma una cola de impresión. Los mismos se irán imprimiendo en el orden en el cual fueron introducidos en la cola.

Otro caso de aplicaciones de colas en computación, es el que se presenta en los sistemas de tiempo compartido. Varios usuarios comparten ciertos recursos, como CPU y memoria de la computadora. Los recursos se asignan a los procesos que están en cola de espera, suponiendo que todos tienen una misma prioridad, en el orden en el cual fueron introducidos a la cola.

#### 6.8 Ejercicios.

1. Utilizando el algoritmo antes expuesto, traducir las siguientes expresiones a notación postfija.
  - $X * (z + w) / (t - v)$
  - $Z - w * y + x^k$
  - $X^k (z - t) * w$
  - $z / (x + y * t)^w$
  - $w * (z / (k - t))^w$
2. Escriba un programa que lea una expresión infija y la traduzca a prefija. El algoritmo es:
  - Primero se tiene que leer la expresión infija.

- El recorrido de la expresión infija será de derecha a izquierda.
  - Si el símbolo a analizar es paréntesis derecho, éste se colocará en tope de pila.
  - Si el símbolo a analizar es paréntesis izquierdo, se imprimirán los símbolos localizados en pila hasta localizar el paréntesis derecho, se elimina el paréntesis derecho de pila y se lee el siguiente símbolo de la expresión infija.
  - Si el símbolo a analizar es un operando, éste se imprime en forma directa.
  - Si el símbolo a analizar es un operador, imprimir lo que está en tope de pila mientras que la pila tenga elementos y la prioridad del operador a analizar sea menor a la prioridad del operador que está en tope de pila. Al no cumplir alguno de los requerimientos antes indicados, se introduce el operador a analizar a tope de pila.
  - Si ya no existen elementos a analizar, se imprime lo que se tenga en pila hasta que se tenga pila vacía.
3. Escriba un programa que invierta los elementos de una cola.
  4. Defina un algoritmo que lea un número entero y que se guarde en la cola uno si es par y se coloque en la cola dos si es impar. Además, que se coloque en la cola tres si es un número primo.

## SEPTIMO CAPÍTULO: Recursividad Directa.

### Resumen.

Este capítulo forma parte de la tercer unidad de competencia dentro de “Recursividad Directa: Definición. Funcionamiento”. La recursividad es una forma muy elegante pero a la vez puede ser la más difícil de comprender o de programar. Varios algoritmos (como algunos algoritmos de ordenación, por ejemplo Quick Sort; algunos algoritmos de programación dinámica, por ejemplo agente viajero; algunos algoritmos que requieren retorno atrás o back tracking, por ejemplo el problema de la N reinas, etc) son recursivos.

Los programas desarrollados en este capítulo fueron realizados por el propio autor y los conceptos vistos en este capítulo se tomaron principalmente de:

(Cairó/Gardati, 2000)

(Loomis, 2013)

(Levin, 2004)

(Serie de Fibonacci)

<http://www.ugr.es/~eaznar/fibo.htm>

El de recursividad para explicar la serie de Fibonacci es dada por el propio autor, así también la definición recursiva tanto del número binario como del determinante.

### 7.1 Definición de recursividad.

El área de la programación es muy amplia y con muchos detalles. En el lenguaje de programación C, así como en otros lenguajes de programación, se puede aplicar una técnica que se le dio el nombre de recursividad por su funcionalidad. La asignación de memoria puede ser estática o dinámica y en un momento dado se puede emplear la combinación de estas dos.

La recursividad es una técnica con la que un problema se resuelve sustituyéndolo por otro problema de la misma forma pero más simple.

Por ejemplo, la función factorial se define como se muestra en el gráfico 7.0:

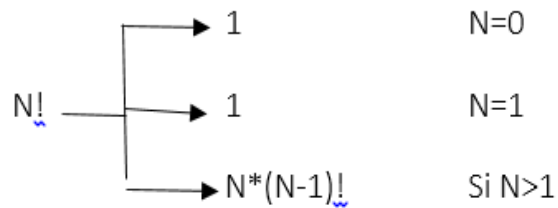


Gráfico 7.0

Siendo ésta definición totalmente recursiva, ya que se vuelve a emplear el mismo concepto cuando  $N > 1$ . Por lo que ciertos problemas se adaptan de manera natural a soluciones recursivas.

## 7.2 Clasificación de funciones recursivas.

Las funciones recursivas se clasifican según se haga la llamada recursiva en:

- Recursividad directa: La función se llama a sí misma
- Recursividad indirecta: La función A llama a la función B, y la función B llama a A.

Según el número de llamadas recursivas generadas en tiempo de ejecución:

- Función recursiva lineal o simple: Se genera una única llamada interna.
- Función recursiva no lineal o múltiple: Se generan dos o más llamadas internas.

Según el punto donde se realice la llamada recursiva, la función recursiva puede ser:

- Final: (Tail recursión): La llamada recursiva es la última instrucción que se produce dentro de la función.
- No final: (Nontail recursive Function): Se realiza alguna operación al volver de la llamada recursiva.

Las funciones recursivas finales suelen ser más eficientes (en la constante multiplicativa en cuanto al tiempo, y sobre todo en cuanto al espacio de memoria) que las no finales. (Algunos compiladores pueden optimizar automáticamente estas funciones pasándolas a iterativas).

Un ejemplo de recursividad final es el algoritmo de Euclides que se puede observar en el gráfico 7.1 para calcular el máximo común divisor de dos números enteros positivos:

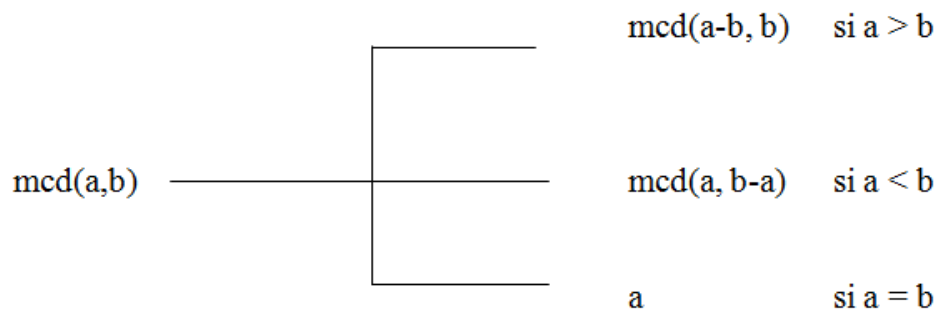


Gráfico 7.1

### 7.3 Diseño de funciones recursivas

- El problema original se puede transformar en otro problema similar más simple
- Tenemos alguna manera directa de solucionar “problemas triviales”

Para que el módulo recursivo sea correcto se debe realizar:

- Un análisis por casos del problema: Existe al menos una condición de terminación en la cual no es necesario una llamada recursiva. Son los casos triviales que se solucionan en forma directa. Ejemplo:

Si  $n=0$  o  $n=1$ , el factorial es 1

- Convergencia de la llamada recursiva: Cada llamada recursiva se realiza con un dato más pequeño, de forma que se llegue a la condición de terminación. Por ejemplo:

$\text{Factorial}(n) = n * \text{Factorial}(n-1)$

- Si las llamadas recursivas funcionan bien, el módulo completo funciona bien: principio de inducción.

$\text{Factorial}(0) = 1$

$\text{Factorial}(1) = 1$

Para  $n > 1$ , si suponemos correcto el cálculo del factorial de  $(n-1)$ ,

$\text{Factorial}(n) = n * \text{Factorial}(n-1)$

Gráficamente, la función factorial quedaría (ver gráfico 7.2):

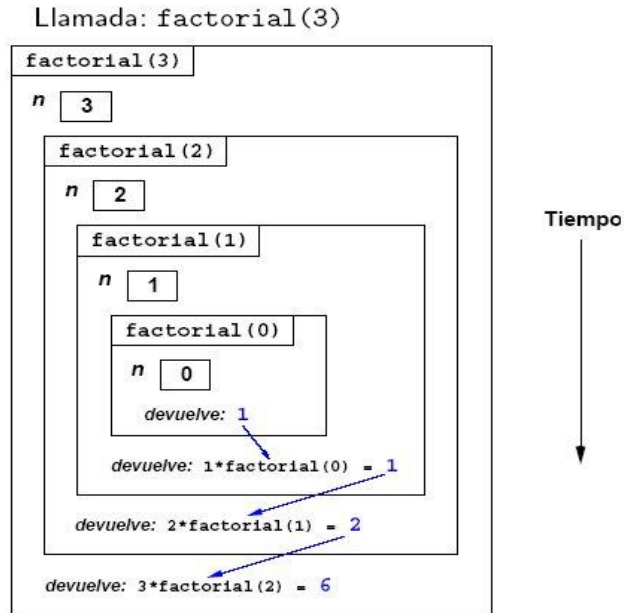


Gráfico 7.2

En código, la función factorial se observa en el programa 7.0

```
#include <stdio.h>
int factorial(int);
main(){
    int i;
    printf("Da el factorial a calcular:");
    scanf("%d",&i);
    printf("Fact(%d)=%d\n",i,factorial(i));
    getchar();
}

int factorial(int j){
    if (j<2)
        return 1;
    return j*factorial(j-1);
}
```

Programa 7.0

Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas así mismo.

#### 7.4 Ventajas e inconvenientes de la recursividad.



Una función recursiva debe ser una manera natural, sencilla, comprensible y elegante del problema. Por ejemplo, dado un número entero no negativo, escribir su codificación en binario (Ver programa 7.1).

```
#include <stdio.h>
void binario(int);
main(){
    int i;
    printf("Da el Numero entero a convertir a binario:");
    scanf("%d",&i);
    printf("Binario(%d)=",i);
    binario(i);
    getchar();
}

void binario(int n){
    if(n<2)
        printf("%d", n);
    else{
        binario(n/2);
        printf ("%d",n%2);
    }
}
```

Programa 7.1

Otro elemento a tomar en cuenta es la facilidad para comprobar y verificar que la solución es correcta (inducción matemática).

En general, las soluciones recursivas son más ineficientes en tiempo y espacio que las versiones iterativas, debido a las llamadas a subprogramas, la creación de variables dinámicas en la pila recursiva y la duplicación de variables. Otra desventaja es que en algunas soluciones recursivas repiten cálculos en forma innecesaria. Por ejemplo, el cálculo del  $n$ -ésimo término de la sucesión de Fibonacci (Ver programa 7.2).

```
#include <stdio.h>
int fibo(int);
main(){
    int i;
    printf("Da el Numero fibonacci a calcular:");
    scanf("%d",&i);
    printf("Fibo(%d)=%d\n",i,fibo(i));
    getchar();
}
```

```

int fibo(int n){
    if(n<2)
        return 1;
    return fibo(n-1) + fibo(n-2);
}

```

Programa 7.2

Fibonacci es una función recursiva no lineal ya que tiene dos llamadas recursivas. Para comprender mejor estas dos llamadas recursivas, y para observar la forma en que se repiten los cálculos, se puede hacer un árbol recursivo como se observa en la figura 7.1:

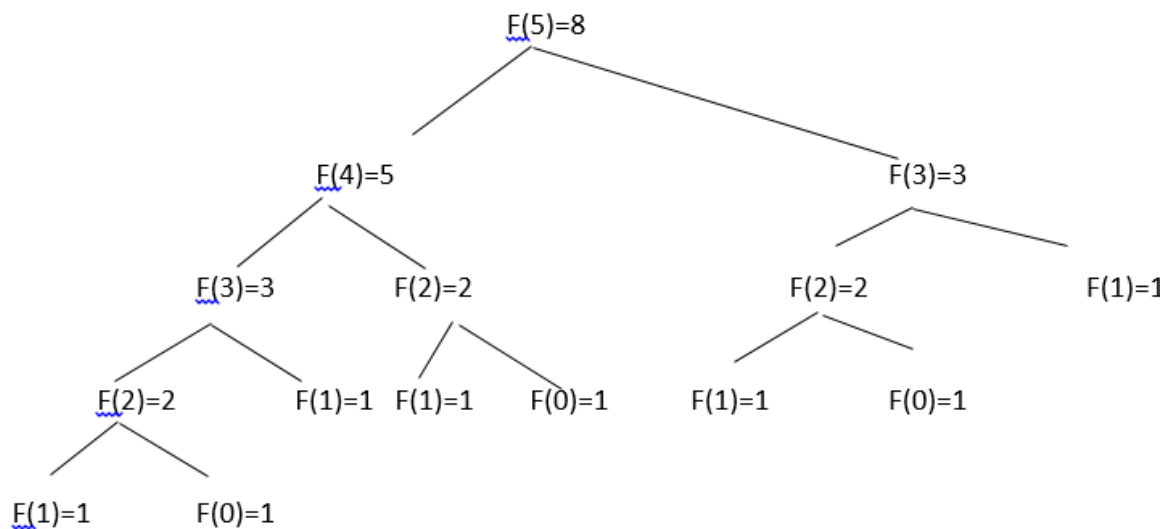


Figura 7.1. Árbol recursivo para el cálculo de Fibonacci de 5.

Para calcular  $F_{200}$ , la función `fib()` se ejecuta en  $2^{138}$  pasos elementales de cómputo. ¿Cuánto tiempo se requiere? Bueno, eso depende de la computadora usada. En este momento, una de las computadoras más veloces en el mundo es la NEC Earth Simulator, con 400 trillones de pasos por segundo. Aún para ésta máquina, `fib(200)` tardará al menos  $2^{92}$  segundos. Esto significa que, si nosotros iniciamos el cómputo hoy, estaría trabajando después de que el sol se torne una estrella roja gigante. Como se observa en la figura 7.1, existen llamadas recursivas en vano. Por ejemplo, para calcular  $F_5$  se tiene que calcular  $F_1$  cinco veces.

Se muestra en lenguaje C un algoritmo mucho más sencillo:

```

#include <stdio.h>
main(){
    int i, n=5, fibn_2, fibn_1, fibn;

    fibn_2=fibn_1=1;
    for ( i =2;i<=n; i++){
        fibn=fibn_2+fibn_1;
        printf ("fibo (%d) =%d\n", i , fibn);
        fibn_2=fibn_1;
        fibn_1=fibn;
    }
    getchar ();
}

```

¿Cuánto tiempo toma el algoritmo? El loop tiene sólo un paso y se ejecuta  $n-1$  veces. Por lo que el algoritmo se considera lineal en  $n$ . De un tiempo exponencial, hemos pasado a un tiempo polinomial, un gran progreso en tiempo de ejecución. Es ahora razonable calcular  $F_{200}$  o aun  $F_{200000}$ .

### 7.5 Cálculo de determinantes en forma recursiva

El cálculo del determinante por menores (ver gráfico 7.3) de una matriz se puede definir en forma recursiva como:

$$\begin{array}{lcl}
 \text{DET}(N, A_N) & \rightarrow & \sum_{j=0}^{N-1} \text{DET}(N-1, A_{N-1}) * a[0][j] * (-1)^j \quad \text{Si } N > 2 \\
 & \rightarrow & a[0][0] * a[1][1] - a[1][0] * a[0][1] \quad \text{Si } N=2 \\
 & \rightarrow & a[0][0] \quad \text{Si } N=1
 \end{array}$$

Gráfico 7.3

Se observará que en la llamada recursiva  $\text{DET}(N-1, A_{N-1})$  los parámetros son  $N-1$ , por lo que se elimina una hilera y una columna a la matriz. En el caso del programa 7.3, se muestra el código que permite calcular el determinante en forma recursiva, en este caso se utiliza la función “subm()” en el cual escoge para su eliminación *la primera* hilera y la *i-ésima* columna (la *i-ésima* columna la indica el comando for localizado en la función “determinate()”).

```

#include <stdio.h>
#include <stdlib.h>
//PROTOTIPOS
int signo(int);

```

```

int determinante(int **,int);
int ** subm(int , int **, int);
main(){
int f,i,j;
int **pm;
printf("Da el tamaño de la matriz=>");
scanf("%d",&f);
pm=(int **)malloc(sizeof(int *)*f);
for (j=0;j<f;j++){
    pm[j]=(int*)malloc(sizeof(int)*f);
for (i=0;i<f;i++){
    for (j=0;j<f;j++){
        printf("a[%d][%d]=",i,j);
        scanf("%d",&pm[i][j]);
    }
}
printf("LA MATRIZ ES:\n");
for (i=0;i<f;i++){
    for (j=0;j<f;j++){
        printf("%d\t",pm[i][j]);
        putchar('\n');
    }
}
printf("\nDeterminante(A)=%d\n",determinante(pm,f));
getchar();
getchar();
}

```

```

int determinante(int ** a, int tam){
if (tam==1)
    return a[0][0];
if (tam==2)
    return a[0][0]*a[1][1]-a[1][0]*a[0][1];
int m=0;
int tam1=tam-1;
for (int i=0;i<tam;i++){
    m=signo(i)*determinante(subm(i,a,tam1),tam1)*a[0][i]+m;
}
return m;
}

```

```

int ** subm(int k, int ** a,int tam){
int ** b;
int i,j;
b=(int **)malloc(sizeof(int *)*tam);
for (j=0;j<tam;j++){
    b[j]=(int*)malloc(sizeof(int)*tam);
}
}

```

```

for (i=0; i<tam;i++){
    int m=0;
    for (j=0; j<tam+1;j++)
        if (j != k){
            b[i][m]=a[i+1][j];
            m++;
        }
}
return b;
}

int signo(int i){
    if (i%2==0)
        return 1;
    return -1;
}

```

Programa 7.3

Cualquier función recursiva se puede transformar en una función iterativa.

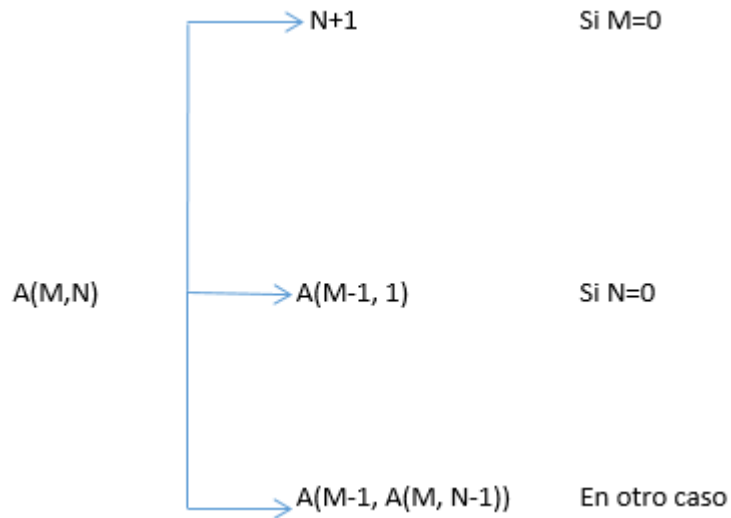
- Ventaja de la función iterativa: Más eficiente en tiempo y espacio.
- Desventaja de la función iterativa: en algunos casos, muy complicada; además, suelen necesitarse estructuras de datos auxiliares.

Si la eficiencia es un parámetro crítico, y la función se va a ejecutar frecuentemente, conviene escribir una solución iterativa. La recursión se puede simular con el uso de pilas para transformar un programa recursivo en iterativo. Las pilas se usan para almacenar los valores de los parámetros del subprograma, los valores de las variables locales, y los resultados de la función.

Ejercicios.

1. Realice un árbol recursivo con Fibonacci(7)
2. Realice un árbol recursivo con la función de Hanoi(4,O,D,A)  
Hanoi(N,Origen, Destino, Auxiliar)  
Si N=1  
Imprime "Mover disco de" Origen "a" Destino  
Sino  
Hanoi(N-1, Origen, Auxiliar, Destino)  
Imprime "Mover disco de" Origen "a" Destino  
Hanoi(N-1, Auxiliar, Destino, Origen)

3. La función de Ackermann se define como:



Realizar un árbol recursivo con  $A(2,2)$

4. El algoritmo de Euclides para el cálculo del máximo común divisor se define como:



Realizar un árbol recursivo para  $MCD(15,4)$  y  $MCD(15,3)$

- Realizar un programa que imprima una palabra al revés (no importando su dimensión) **sin el uso de arreglos o memoria dinámica.**
- Hacer un árbol recursivo para la función del determinante suponiendo que se tiene una matriz de  $4 \times 4$ .

## OCTAVO CAPÍTULO: ÁRBOLES BINARIOS.

### Resumen

Este capítulo es parte de la Tercer Unidad de Competencias dentro de “Árboles: Usos. Características. Árboles Binarios: Representación. Operaciones. Recorrido (Preorden, Orden, Postorden). Árboles Binarios de Expresiones: Características. Evaluación de expresiones aritméticas. Árboles Binarios de Búsqueda: Características. Operaciones (Inserción, Eliminación, Búsqueda)”.

Los conceptos fueron tomados de los siguientes libros:

(Dasgupta, Papadimitriou, & Vazirani, 2008)

(Cairó/Gardati, 2000)

(Horowitz, 1978)

El pseudocódigo del árbol binario de búsqueda fue tomado de: (Cairó/Gardati, 2000). La programación del algoritmo es de propia autoría.

### 8.1 Introducción.

Los árboles representan las estructuras no lineales y dinámicas más importantes en cómputo. Un árbol binario es una estructura de datos en la cual cada nodo siempre tiene un hijo izquierdo y un hijo derecho. No puede tener más de dos hijos. Si algún hijo tiene como referencia a NULL, es decir que no almacena ningún dato, entonces éste es llamado un nodo externo o nodo hoja. En el caso contrario el hijo es llamado un nodo interno.

En teoría de grafos, se usa la siguiente definición: “Un árbol binario es un grafo conexo, acíclico y no dirigido tal que el grado de cada vértice no es mayor a 3”. De esta forma sólo existe un camino entre un par de nodos.

Un árbol binario tiene múltiples aplicaciones. Se los puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos de un proceso, para representar la historia de un campeonato de tenis, para representar un árbol genealógico y para representar expresiones algebraicas construidas con operadores binarios. Esto sólo para citar algunos de sus múltiples usos.

Existen dos formas tradicionales de representar un árbol binario en memoria:

1. Por medio de datos tipo apuntador. También conocidos como variables dinámicas.
2. Por medio de arreglos.

Sin embargo, en este capítulo se explicará la primer forma, puesto que es la más natural para tratar este tipo de estructuras.

Los nodos de árboles binarios serán representados como registros, que contendrán como mínimo tres campos. En un campo se almacenará la información del nodo. Los otros dos restantes se utilizarán para apuntar a los subárboles izquierdo y derecho respectivamente del nodo en cuestión. Dado el siguiente nodo T:

T

IZQ	INFO	DER
-----	------	-----

Éste tiene los siguientes campos:

IZQ: Campo donde se almacena la dirección del subárbol izquierdo del nodo T.

INFO: Campo donde se almacena la información de interés del nodo. Normalmente en éste campo y en el transcurso de los apuntes se almacenará un valor simple: número o carácter. Sin embargo, en la práctica es común almacenar en este campo registros, arreglos y conjuntos.

DER: Campo donde se almacenará la dirección del subárbol derecho del nodo T.

## 8.2 Árboles binarios de expresiones

Una expresión algebraica donde se tengan operadores binarios se puede representar por medio de un árbol binario. En un árbol binario se puede guardar perfectamente la prioridad relativa de una expresión algebraica. Los operadores más cercanos a la raíz son operadores que tienen menor prioridad relativa y los que se localizan más lejos de la raíz tendrán una mayor prioridad relativa.

Ejemplo:

(En este ejemplo, y para el resto de los apuntes, el signo “^” indicará potencia.)

Se tiene la siguiente expresión matemática:  $a + b * c / d + e ^ (f + g)$

Las prioridades relativas se muestran en el gráfico 8.1:

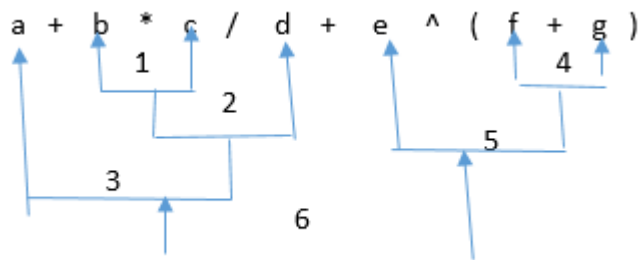


Gráfico 8.1

Con las prioridades relativas marcadas se puede hacer un árbol binario de la expresión antes indicado (Ver gráfico 8.2):



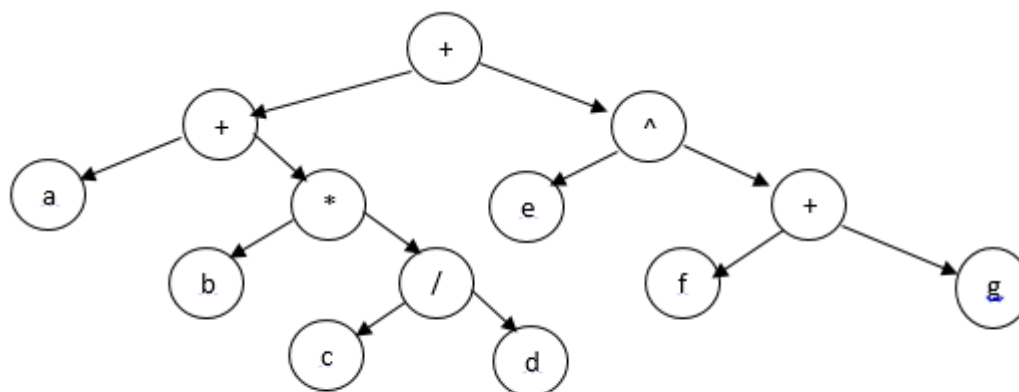


Gráfico 8.2

Nota: El crear un árbol de expresiones matemáticas, como el que se muestra en la gráfica 8.2 requiere un autómata tipo pila, y tal detalle se observa plenamente en un curso que trate sobre Compiladores.

### 8.3 Recorrido de un árbol binario:

Una de las operaciones más importantes a realizar en un árbol binario es el recorrido del mismo. Recorrer significa visitar los nodos del árbol en forma sistemática; de tal manera que todos los nodos del mismo sean visitados una sola vez a la vez. Existen tres formas diferentes de efectuar el recorrido y todas de ellas de naturaleza recursiva, estas son:

#### a) Recorrido en preorden

- Visitar la raíz
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho

Al realizar el recorrido en preorden del gráfico 8.2 se obtiene la notación polaca en prefijo: ++a\*b/cd^e+fg

#### b) Recorrido en inorden

- Recorrer el subárbol izquierdo
- Visitar la raíz
- Recorrer el subárbol derecho

Al realizar el recorrido en inorden se produce la notación convencional.

#### Recorrido en postorden

- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho
- Visitar la raíz.

Al realizar el recorrido en postorden se produce la notación polaca en postfijo.  
El recorrido del árbol en el gráfico 8.2 se obtiene:  $abcd/*+efg+^+$

#### 8.4 Árboles binarios de búsqueda

El árbol binario de búsqueda es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación.

Un árbol binario de búsqueda se define de la siguiente forma: "Para todo nodo  $T$  del árbol debe cumplirse:

*Que todos los valores de los nodos del subárbol izquierdo de  $T$  deben ser menores o iguales al valor del nodo  $T$ . En forma similar, todos los nodos del subárbol derecho de  $T$  deben ser mayores o iguales al valor del nodo  $T$ ".*

##### 8.4.1 La inserción de un árbol binario de búsqueda

La inserción es una operación que se puede realizar eficientemente en un árbol binario de búsqueda. Los pasos a realizarse para insertar un elemento en un árbol binario de búsqueda son:

1. Debe compararse la clave a insertar con la raíz del árbol. Si es mayor, debe avanzarse hacia el subárbol derecho. Si es menor, debe avanzarse hacia el subárbol izquierdo.
2. Repetir el paso 1 hasta que se cumpla alguna de las siguientes condiciones:
  - 2.1 El subárbol derecho es igual a vacío, o el subárbol izquierdo es igual a vacío, en cuyo caso se procederá a insertar el elemento en el lugar que le corresponde.
  - 2.2 La clave que quiere insertarse es igual a la raíz del árbol; en cuyo caso no se realiza la inserción.

##### 8.4.2 El borrado en un árbol binario de búsqueda.

La operación de borrado es un poco más complicada que la de inserción. Ésta consiste en eliminar un nodo del árbol sin violar los principios que definen justamente a un árbol binario de búsqueda. Se deben distinguir los siguientes casos:

1. Si el elemento a borrar es terminal u hoja, simplemente se suprime y su nodo padre debe de apuntar a NULL.
2. Si el elemento a borrar tiene un solo descendiente, entonces tiene que sustituirse ese descendiente.
3. Si el elemento a borrar tiene los dos descendientes, entonces se tiene que sustituir por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Además, debemos recordar que antes de eliminar un nodo, debemos localizarlo en el árbol. Para esto, se utilizará el algoritmo de búsqueda presentado anteriormente.

#### 8.4.3 Recorrido en un árbol binario de búsqueda.

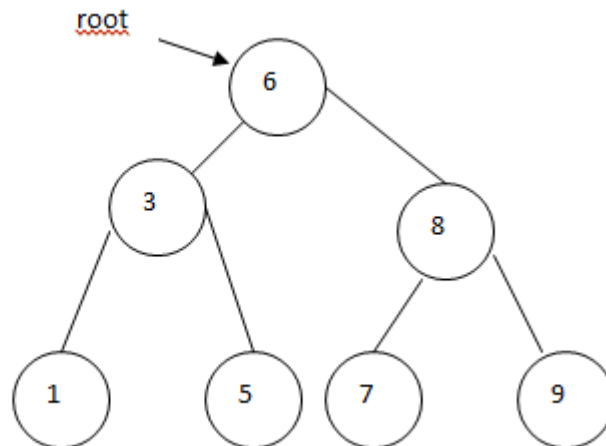
Una de las operaciones más importantes a realizar en un árbol binario de búsqueda es el recorrido de los mismos, recorrer significa visitar los nodos del árbol en forma sistemática, de tal manera que todos los nodos del mismo sean visitados en una sola vez. Existen 3 formas diferentes de efectuar el recorrido y todas ellas de naturaleza recursiva, estas son:

Recorrido en preorden: en el que se procesa el nodo y después se procesan recursivamente sus hijos.

Recorrido en postorden: Donde el nodo dado se procesa después de haber procesado recursivamente a sus hijos.

Recorrido en inorden: en este se procesa recursivamente el hijo izquierdo, luego se procesa el nodo actual y finalmente se procesa recursivamente el hijo derecho.

Ejemplificando con el siguiente árbol se tiene:



Recorrido en preorden:

6-3-1-5-8-7-9

Recorrido en postorden:

1-5-3-7-9-8-6

Recorrido en inorden:

1-3-5-6-7-8-9

La programación de un árbol binario de búsqueda se muestra en el programa 8.1:

```
#include <alloc.h>
struct nodo{
    int dato;
    nodo * izq;
    nodo * der;
};

/*PROTOTIPOS
void * crear (void *);
void mostrar(void *);
void * eliminar(void *);
void preorden(void *);
void postorden(void *);
void inorden(void *);
int menu();

main(){
void * root=NULL;
int x;
do{
    x=menu();
    switch(x){
        case 1:root=crear(root);break;
        case 2:mostrar(root);break;
        case 3:root=eliminar(root);break;
    }
}while (x>0 && x<4);
return 0;
}

void mostrar(void *r){
int x;
do{
    printf("\n\tSUBMENU MOSTRAR\n");
    printf("1. MOSTRAR EL ARBOL EN PREORDEN\n");
    printf("2. MOSTRAR EL ARBOL EN POSTORDEN\n");
    printf("3. MOSTRAR EL ARBOL EN INORDEN\n");
    printf("4. SALIR\n");
    printf("Da la opcion->");
    scanf("%d",&x);
```

```

}while(x>4 || x < 1);

if (r==NULL)
    printf("ARBOL VACIO\n");
else
    switch(x){
        case 1:preorden(r);break;
        case 2:postorden(r);break;
        case 3:inorden(r);break;
    }
}

int menu(){
int x;
do{
    printf("\n\tMENU\n");
    printf("1. INSERTAR UN ELEMENTO\n");
    printf("2. MOSTRAR EL ARBOL\n");
    printf("3. ELIMINAR UN ELEMENTO DEL ARBOL\n");
    printf("4. SALIR\n");
    printf("Da la opcion->");
    scanf("%d",&x);
}while(x>4 || x < 1);
return(x);
}

void * eliminar(void * r){
/*Se emplea una variable del tipo bandera para:
    Si band=1, no se localizó el dato en el árbol
    Si band =2, se localizó el dato en el árbol
*/
if (r==NULL)
    printf("ARBOL VACIO\n");
else{
    printf("DA EL VALOR DEL DATO A ELIMINAR=>");
    int x,band=0;
    scanf("%d",&x);
    nodo *aux,*q=(nodo *)r;
    aux=NULL;
    do
        if(q->dato<x)
            if(q->der!=NULL){
                aux=q;
                q=q->der;
            }
        else if(q->dato==x){
            band=1;
            if(q->der!=NULL){
                aux=q->der;
            }
            if(q->izq!=NULL){
                aux=q->izq;
            }
            if(aux==NULL){
                printf("No se localizó el dato en el árbol\n");
            }
        }
        else if(q->dato>x){
            band=2;
            q=q->izq;
        }
    }while(q!=NULL);
    if(band==1){
        printf("No se localizó el dato en el árbol\n");
    }
    else if(band==2){
        printf("Se localizó el dato en el árbol\n");
    }
}
}

```

```

        }
        else
            band=1;
    else if(q->dato>x)
        if(q->izq!=NULL){
            aux=q;
            q=q->izq;
        }
        else
            band=1;
    else band=2;
while(band==0);
if(band==2){
    printf("DATO A ELIMINAR=>%d\n",q->dato);
    if(aux!=NULL)
        printf("VALOR DEL DATO EN EL AUXILIAR=>%d\n",aux->dato);
    else
        printf("DATO LOCALIZADO EN LA RAIZ\n");
    nodo * d;
    if (q->izq==NULL && q->der==NULL){
        printf("EL DATO SE LOCALIZO EN UN NODO HOJA\n");
        if (r==q){
            printf("Y ES NODO RAIZ UNICO\n");
            r=NULL;
        }
        else if(aux->izq==q)
            aux->izq=NULL;
        else if(aux->der==q)
            aux->der=NULL;
        d=q;
    }
    else if (q->izq==NULL && q->der!=NULL){
        // El dato a eliminar solo tiene ramificación derecha
        d=q;
        q=q->der;
        if (r==d)
            r=d->der;
        else if(aux->der==d)
            aux->der=d->der;
        else
            aux->izq=d->der;
    }
    else if (q->izq!=NULL && q->der==NULL){
        // El dato a eliminar solo tiene ramificación izquierda

```

```

        d=q;
        q=q->izq;
        if(r==d)
            r=d->izq;
        else if(aux->der==d)
            aux->der=d->izq;
        else
            aux->izq=d->izq;
    }
    else{
        // El dato a eliminar tiene las dos ramificaciones
        d=q->izq;
        if(d->der==NULL){
            q->dato=d->dato;
            q->izq=d->izq;
        }
        else{
            do{
                aux=d;
                d=d->der;
            }while(d->der!=NULL);
            if (d->izq!=NULL)
                aux->der=d->izq;
            else
                aux->der=NULL;
            q->dato=d->dato;
        }
    }
    free(d);
} //if (q->dato==x)
else
    printf("DATO NO LOCALIZADO\n");
} //else de arbol no vacio
return(r);
}

void * crear(void * p){
/*Se utiliza una variable tipo bandera para:
    Si band=1, el valor a insertarse está repetido
    Si band=2, Se llegó a una hoja donde se insertará el nuevo elemento
    Si band=3, el árbol está vacío
*/
int x,band=0;
nodo * q, * aux;

```

```

printf("Da un valor entero=>");
scanf("%d",&x);
aux=(nodo *)p;
if (aux==NULL)
    band=3;
else
    do
        if (aux->dato==x)
            band=1;//valor repetido
        else if (aux->dato>x){
            if(aux->izq!=NULL)
                aux=aux->izq;
            else
                band=2;
        }
        else{
            if(aux->der!=NULL)
                aux=aux->der;
            else
                band=2;
        }
    while (band<1);
if (band>1){
    q=(nodo*)malloc(sizeof(nodo));
    q->der=q->izq=NULL;
    q->dato=x;
    if (band==3)
        p=q;
    else if (aux->dato<x)
        aux->der=q;
    else
        aux->izq=q;
}
else
    printf("VALOR REPETIDO\n");

return(p);
}

void preorden(void * r){
    nodo *w;
    w=(nodo*)r;
    if (r!=NULL){
        printf("%d..",w->dato);
    }
}

```



```

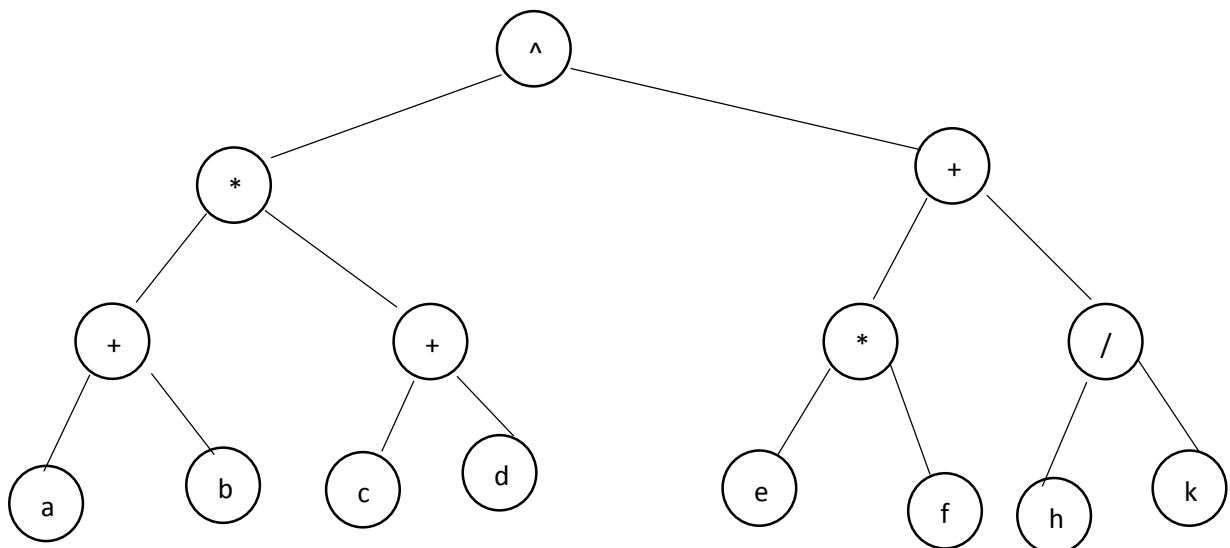
        preorden(w->izq);
        preorden(w->der);
    }
}
void postorden(void * r){
    nodo *w;
    w=(nodo*)r;
    if (r!=NULL){
        postorden(w->izq);
        postorden(w->der);
        printf("%d...",w->dato);
    }
}
void inorden(void * r){
    nodo *w;
    w=(nodo*)r;
    if(w!=NULL){
        inorden(w->izq);
        printf("%d...",w->dato);
        inorden(w->der);
    }
}
}

```

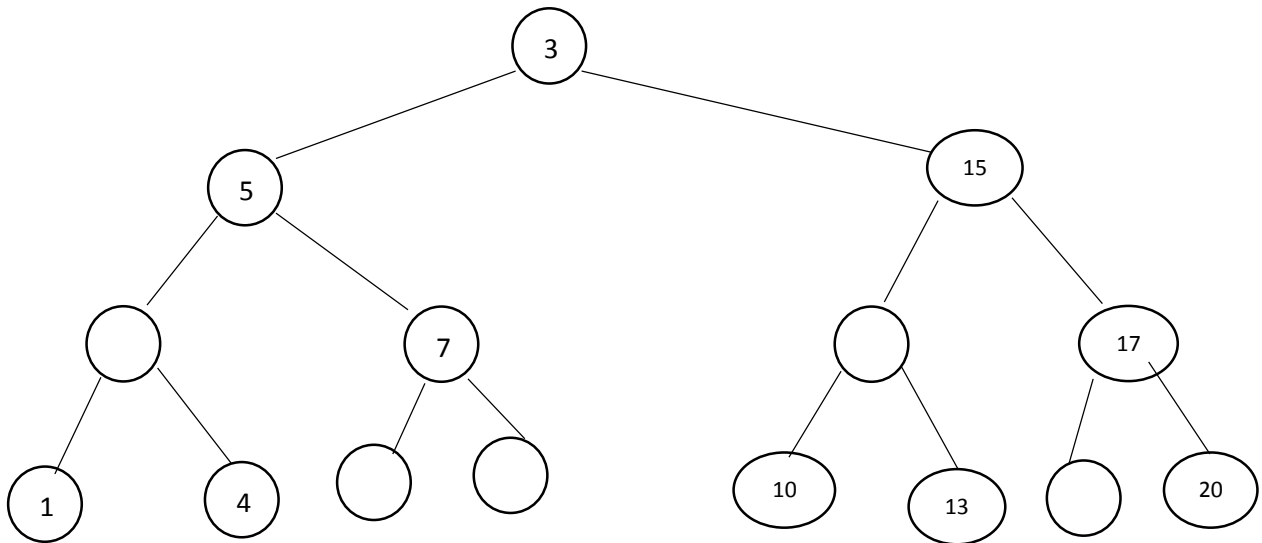
Programa 8.1

Ejercicios:

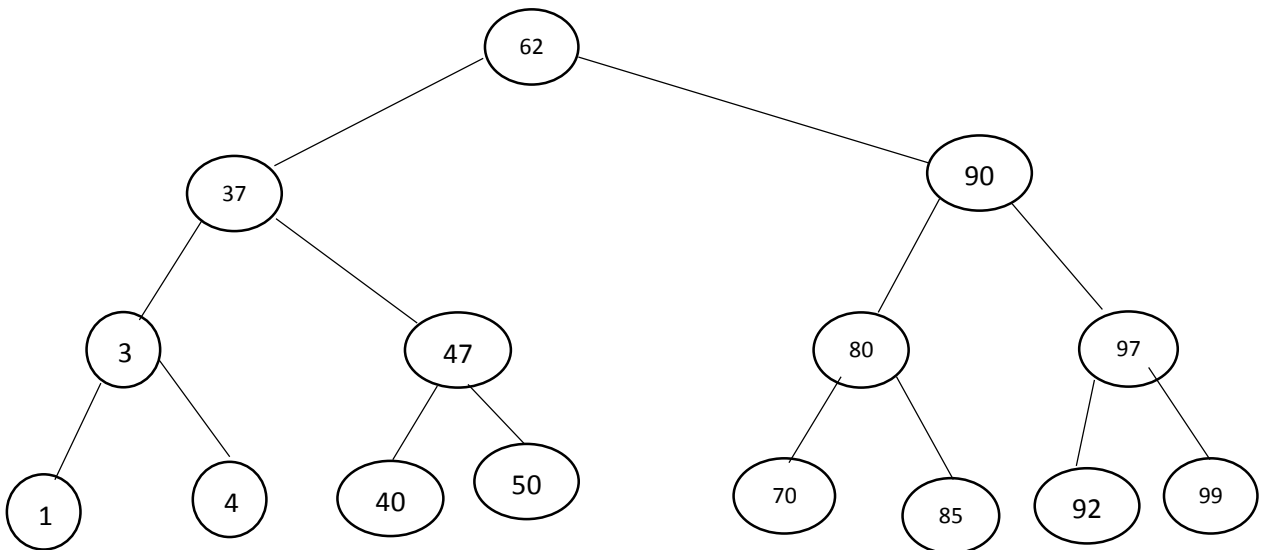
1. Dado el siguiente árbol binario de expresiones, realizar el recorrido en preorden, inorden y postorden.



2. Dado el siguiente árbol binario de búsqueda, complete los nodos en blanco de tal forma que no se violen los principios que definen un árbol binario de búsqueda.



3. Dado el siguiente árbol de búsqueda, elimine las claves 47-37-62-90. En cada eliminación muestre el árbol resultante.



4. Escriba un programa para calcular cuántos nodos tiene un árbol binario, cuál es el máximo valor y cuál es el promedio de los nodos.

5. Escriba un procedimiento que realice lo siguiente:
  - a. Imprimir sólo las hojas de un nodo binario.
  - b. Imprimir sólo los nodos internos de un árbol binario.
6. Escriba un procedimiento que efectúe los recorridos en preorden, inorden y postorden en forma iterativa en lugar de recursiva (deberá utilizar una pila).
7. Escriba un programa que cargue los nodos de un árbol en un arreglo unidimensional. Cuide que se mantenga la relación padre-hijo entre los nodos.

## NOVENO CAPÍTULO: GRAFOS

### Resumen.

En este capítulo se desarrolla en forma extensa la cuarta unidad de competencias en la cual solicita: “Grafos: Características. Tipos. Representación y construcción. Operaciones. *“Grafos Dirigidos*: Algoritmos para la obtención del camino más corto. *Grafos No dirigidos*: Algoritmos para la obtención de costo mínimo”. Una gran cantidad de problemas de la vida real como problemas que aparecen en la logística, la robótica, la genética, la sociología, el diseño de redes y el cálculo de rutas óptimas se representan por medio de un grafo. Y la representación matemática de un grafo se ve plasmada por medio de una matriz.

En la primer parte de este capítulo se comenta la teoría básica de los grafos y la mayoría de su texto se tomó del (Cairó/Gardati, 2000). Los ejemplos y el pseudocódigo fueron tomados del (Horowitz, 1978). Algunas ideas básicas fueron tomadas de (Dasgupta, Papadimitriou, & Vazirani, 2008).

### 9.1 Introducción

En el capítulo 8 se estudiaron las estructuras de datos tipo árboles, en las cuales cada nodo o elemento puede tener como máximo un nodo precedente o raíz. Sin embargo, existen situaciones en las que la información a representar no responde a una estructura de este tipo sino que necesita una mayor flexibilidad para otro tipo de relaciones entre los datos o componentes de la información. Por ello dedicaremos este capítulo al estudio de los grafos: estructuras de datos que permiten representar diferentes tipos de relaciones entre los objetos.

En un grafo se distinguen básicamente dos elementos: los nodos, mejor conocidos como vértices y los arcos, llamados aristas, que conectan un vértice con otro. Los vértices almacenan información y las aristas representan relaciones entre dicha información.

Considere, por ejemplo, el grafo de la figura 9.1 en la que se representan algunas de las principales capitales sudamericanas y la conexión entre ellas. En este ejemplo los vértices representan las ciudades, mientras que las aristas representan a las carreteras o algún otro medio de conexión entre ellas.

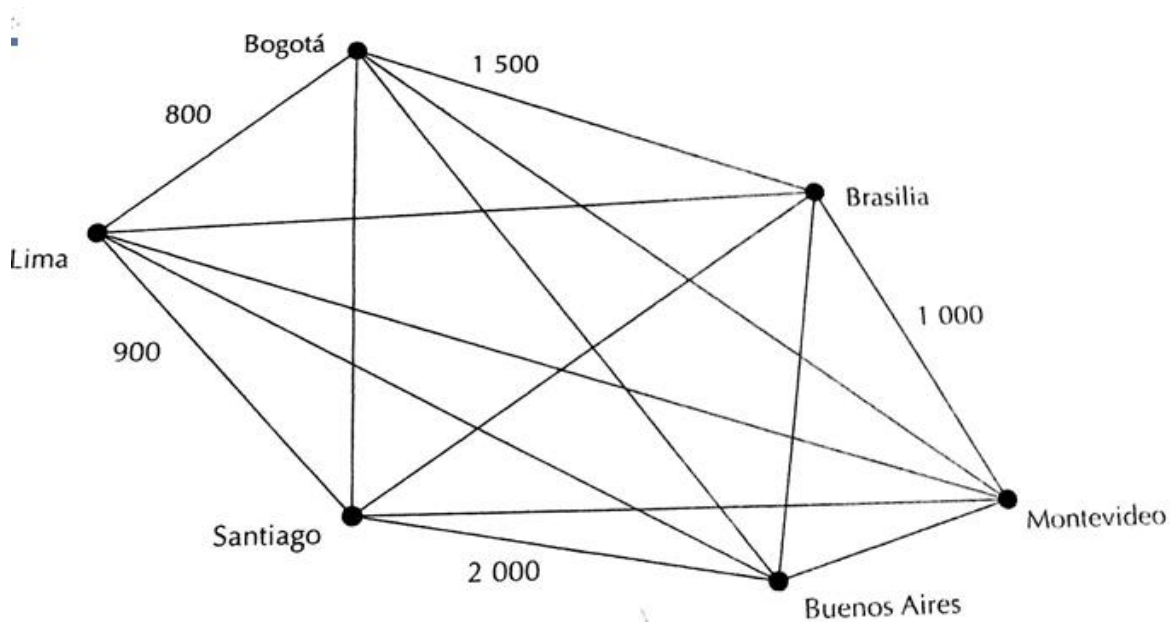


Figura 9.1

Algunas aristas están etiquetadas, y en este caso el valor que aparece en ellas representa la distancia que hay que recorrer para llegar de una ciudad a otra. En general, una etiqueta en la arista que une, por ejemplo, los vértices  $i$  y  $j$  se usan para representar el costo de ir del vértice  $i$  al vértice  $j$ .

## 9.2 Definición de grafo.

Un grafo  $G$  consta de dos conjuntos:  $V(G)$  y  $A(G)$ . El primero de ellos está formado por elementos llamados nodos o vértices, mientras que el segundo conjunto está formado por arcos o aristas. Por lo tanto, podemos denotar a un grafo como:

$$G = (V, A)$$

Dónde:  $V$  simboliza el conjunto de vértices y  $A$  el conjunto de aristas. Si no se hace ninguna especificación, los conjuntos  $V$  y  $A$  son finitos.

Cada arista está identificada por un único par de nodos del conjunto de vértices, el cual puede estar o no ordenado. Una arista que va del vértice  $u$  al vértice  $v$  se denota por medio de la expresión:  $a = (u, v)$ , donde  $u$  y  $v$  son vértices adyacentes y los extremos de  $a$ . En este caso,  $u$  y  $v$  están conectados por  $a$  y se dice que  $a$  es incidente en  $u$  y  $v$ .

Conceptos importantes de grafos.

A continuación se presentan algunos de los conceptos más importantes relacionados con la teoría de grafos.

- Grado de un nodo: El grado de un nodo  $v$ , escrito como  $\text{grad}(v)$ , es el número de aristas que contiene a  $v$ , es decir, que apuntan hacia él. Si el  $\text{grad}(v)=0$  ( $v$  no tiene aristas), se dice que  $v$  es un nodo aislado.
- Lazo o bucle: Un lazo o bucle es una arista que conecta a un nodo consigo mismo. Es decir:  $a = (u, u)$ .
- Camino: Un camino  $P$  de longitud  $n$  desde un vértice  $v$  a un vértice  $w$  se define como la secuencia de  $n$  vértices que se debe seguir para llegar del nodo origen al nodo destino.

$$P = (v_1, \dots, v_n)$$

De tal modo que:  $v = v_i$ ;  $v_i$  es adyacente a  $v_{i+1}$  para  $i=1, 2, \dots, n-1$ ; y  $w = v_n$ .

- Camino cerrado: El camino  $P$  es cerrado si el primer y último nodo son iguales, es decir, si  $v_0 = v_n$ .
- Camino simple: El camino es simple si todos sus nodos son distintos, con excepción del primero y del último, que pueden ser iguales. Es decir,  $P$  es simple si  $v_0, v_1, \dots, v_{n-1}$  son distintos.
- Ciclo: Un ciclo es un camino simple cerrado de longitud 3 o mayor. Un ciclo de longitud  $k$  se llama  $k$ -ciclo.
- Grafo conexo: Se dice que un grafo es conexo si existe un camino simple entre dos de sus nodos cualesquiera.
- Grafo árbol: Se dice que un grafo  $G$  es del tipo árbol o árbol libre, si  $G$  es un grafo conexo sin ciclos.
- Grafo completo: Se dice que un grafo es completo si cada vértice  $v$  de  $G$  es adyacente a todos los demás vértices de  $G$ . Un grafo completa de  $n$  vértices tendrán  $n(n-1)/2$  aristas.
- Grafo etiquetado: Se dice que un grafo  $G$  está etiquetado si sus aristas tienen asignado un valor. Si cada arista  $a$  tiene un valor numérico no negativo  $u(a)$ , llamado peso o longitud de  $a$ , se dice que  $G$  tiene peso. En este caso, cada camino de  $P$  de  $G$  tendrá asociado un peso o longitud que será la suma de los pesos de las aristas que forman el camino  $P$ .
- Multígrafos: Un grafo se denomina multígrafo si al menos dos de sus vértices están conectados por dos aristas. En este caso las aristas reciben el nombre de aristas múltiples o paralelas.
- Subgrafos: Dada la gráfica  $G = (V, A)$ ,  $G' = (V', A')$  se denomina subgrafo de  $G$  si:  $V' \neq \emptyset$ ,  $V' \subseteq V$  y  $A' \subseteq A$ , donde cada arista de  $A'$  es incidente con vértices de  $V'$ .

Grafos dirigidos.

En esta sección se tratara un tipo especial de grafos llamado grafos dirigidos. Además de su definición y su representación, se presentarán dos algoritmos usados para el cálculo de caminos (uno con grafos no dirigidos y el otro con grafos dirigidos). Es importante mencionar que existen una gran cantidad de problemas de la vida real que son de difícil solución, y que sin embargo se pueden resolver –fácilmente, en algunos casos – aplicando la teoría de grafos.

### Definición

Un grafo dirigido se caracteriza porque sus aristas tienen asociada una dirección. Los vértices se utilizan para representar información, mientras que las aristas representan una relación con dirección o jerarquía entre los vértices. Una posible aplicación de este tipo de grafos puede ser la representación de ciudades en los vértices, y la duración de los vuelos en las aristas, Asumiendo que el tiempo necesario para ir de la ciudad  $C_1$  a la ciudad  $C_2$  no es el mismo que el requerido para ir de la ciudad  $C_2$  a la ciudad  $C_1$ . A continuación se define formalmente el concepto de grafos dirigidos.

Una grafo dirigido  $G$ , también llamada digrafo, se caracteriza Porque cada arista  $a$  tiene una dirección asignada; es decir, cada arista está asociada a un par ordenado  $(u, v)$  de nodos de  $G$ . Una arista dirigida  $a = (u, v)$  se llama arco, Y generalmente se expresa como  $u \rightarrow v$ . Para las aristas de las di gráficas se aplica la siguiente terminología:

- a)  $a$  empieza en  $u$  y termina en  $v$ .
- b)  $u$  es el origen o punto inicial de  $a$ , y  $v$  es el destino o punto terminal de  $a$ .
- c)  $u$  es un predecesor de  $v$  y  $v$  es un sucesor o vecino de  $u$ .
- d)  $u$  es adyacente hacia  $v$  y  $v$  es adyacente desde  $u$ .

### Representación de grafos dirigidos.

Los grafos no existen como una herramienta para la representación de datos disponibles en los lenguajes de programación; por lo tanto, para representar los grafos dirigidos se pueden usar otras estructuras de datos de manera auxiliar. Existen varias opciones para realizarlo; la elección más adecuada depende del uso que se le vaya a dar a la información almacenada en los vértices y en las aristas. Las representaciones más utilizadas son las matrices. La matriz de adyacencia es una matriz booleana, es decir, una matriz formada por ceros y unos. Para generar la matriz de adyacencia correspondiente a un grafo, se le dan un orden arbitrario a los vértices del grafo, y si le asigna a los renglones y a las columnas de una matriz el mismo orden. Un elemento de la matriz será 1 si los vértices correspondientes al renglón y a la columna están unidos por una arista (son adyacentes), y será 0 en caso contrario.

Sí  $G=(V,A)$  y  $V=\{1,2,3,...,n\}$ , la matriz de adyacencia que representa la gráfica tiene  $n \times n$  elementos donde  $M[i, j](1 \leq i \leq n \text{ y } 1 \leq j \leq n)$  es 1 sólo si existe un arco que vaya del nodo  $i$  al nodo  $j$ , y es 0 en otro caso cualquiera. Una ventaja de las matrices de adyacencia es que el tiempo de acceso al elementos requeridos es independiente del tamaño de  $V$  y  $A$ ; el tiempo de buscar “a” será del orden de  $O(n)$ . Sin embargo, su principal desventaja es que requiere un espacio de almacenamiento de  $n^2$  posiciones, aunque el número de arcos de  $G$  no sobrepase ese número. La matriz de adyacencia es útil en los algoritmos en los cuales se necesita conocer si existe una arista entre dos nodo estados.

Una variante de la matriz de adyacencia es la matriz de adyacencia etiquetada, en donde  $M[i, j]$ , es la etiqueta de  $a = (i, j)$ . Si la arista no existe, entonces el valor de  $M[i, j]$  será cero.

### 9.3 Árbol de expansión mínima (Minimun Spanning Tree)

Definición: Sea  $G(V, E)$  un grafo no direccionado. Un subgrafo  $T(V, E')$  es un árbol de extensión de  $G$  si y sólo si  $T$  es un árbol (Ver fig. 9.2).

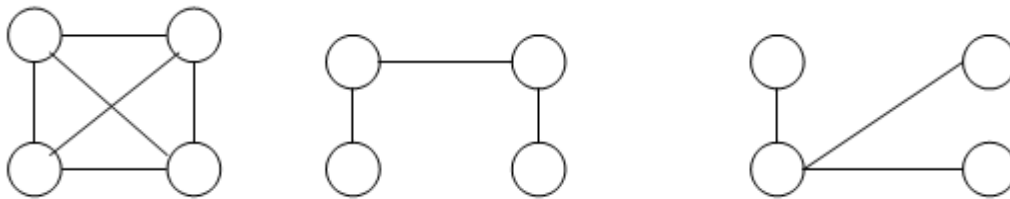


Fig. 9.2 Una subgráfica.

Si los nodos representan ciudades y el segmento que las une representa una posible unión entre ambas ciudades, entonces el mínimo número de segmentos para unir las  $n$  ciudades es  $n-1$ . El árbol de extensión representa todas las posibles combinaciones.

En una situación práctica, los segmentos tendrán pesos asignados. Estos pesos pueden representar costos de construcción, distancias, etc. Ahora, dado un peso, lo que se desea es conectarse a todos los nodos con el mínimo costo. En cualquier caso, los segmentos seleccionados formarán un árbol (suponiendo todos los costos positivos). Lo interesante será localizar un árbol de extensión en  $G$  con el costo mínimo.

Un método Greedy (voraz) para obtener un mínimo costo será ir edificando el árbol segmento por segmento. El siguiente segmento a escoger será aquel en que se minimice el incremento en costos.

Si  $A$  es el conjunto de segmentos seleccionados hasta el momento, entonces  $A$  forma un árbol. El siguiente segmento  $(u, v)$  a ser incluido en  $A$  es un segmento con costo mínimo no



en A con la propiedad de que  $A \cup \{u, v\}$  también es un árbol. Este algoritmo se conoce como el algoritmo de Prim.

En la figura 9.3 se muestra la forma en que trabaja el método PRIM. El árbol de expansión tiene un costo de 105.

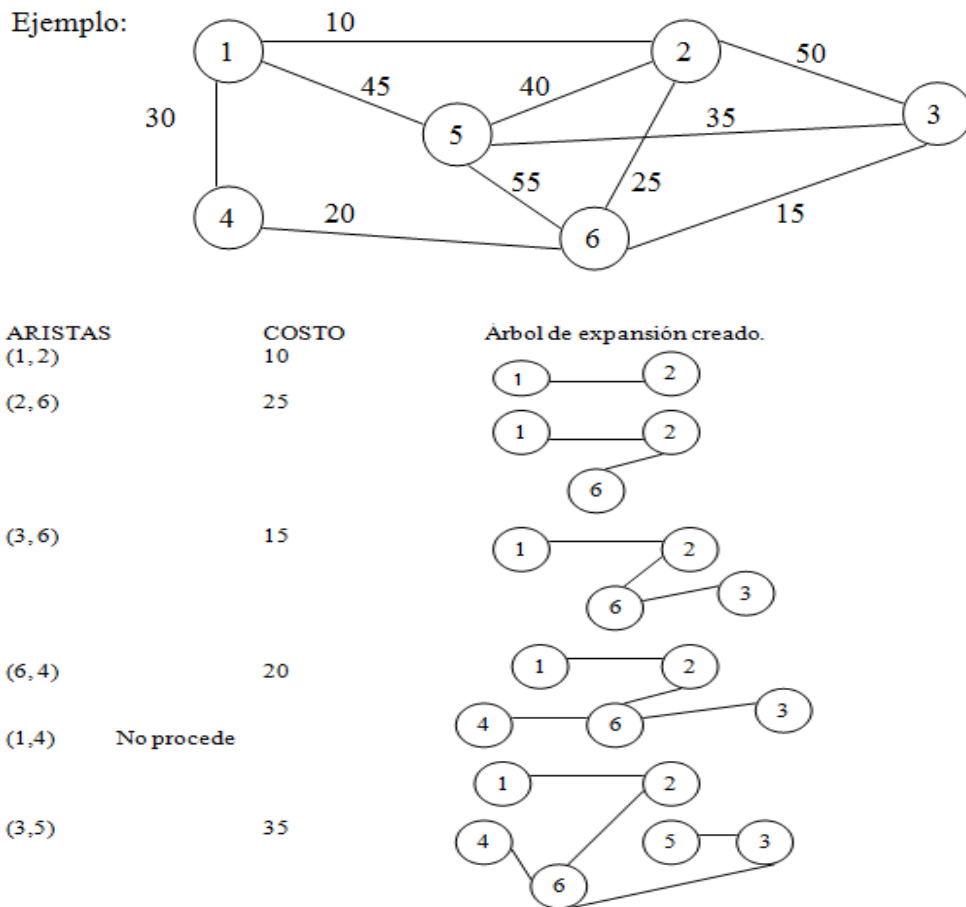


Fig. 9.3 Solución paso a paso del árbol de expansión mínima.

El algoritmo inicia con un árbol que incluye sólo un borde con costo mínimo de G. Entonces, las aristas serán adicionadas al árbol una por una. La siguiente arista  $(i, j)$  a ser adicionada es tal que el vértice  $i$  se encuentra incluido en el árbol y  $j$  es el vértice aún no incluido y el  $\text{COSTO}(i, j)$  es mínimo sobre todas las aristas  $(k, l)$  donde el vértice  $k$  es parte del árbol y el vértice  $l$  no se encuentra en el árbol. En orden de determinar este vértice  $(i, j)$  en forma eficiente, nosotros asociamos por cada vértice  $j$  aún no incluido en el árbol un valor  $\text{NEAR}(j)$ .  $\text{NEAR}(j)$  es un vértice en el árbol tal que  $\text{COSTO}(j, \text{NEAR}(j))$  es mínimo sobre todas las elecciones para  $\text{NEAR}(j)$ . Se define  $\text{NEAR}(j)=0$  para todos los vértices  $j$  que se encuentran en el árbol. La siguiente arista a incluir es definida por el vértice  $j$  tal que  $\text{NEAR}(j) \neq 0$  ( $j$  no se encuentra aún en el árbol) y  $\text{COSTO}(j, \text{NEAR}(j))$  es mínimo. Ver programa 9.1.

```

#include <stdio.h>
#define TAM    6
main(){
    //costo(n,n) es la matriz de costos del trayecto del grafo
    //El costo(i,i) es infinito, el costo(i,j), donde i!=j, es positivo.
    //El trayecto se guarda en el arreglo T(n,2)
    //El costo final se asigna a minicost.
    //Se requiere un vector que indique el nodo más cercano al nodo j, ese vector
    //NEAR se guarda en la variable ne.
    Int
cost[TAM][TAM]={999,10,999,30,45,999},{10,999,50,999,40,25},{999,50,999,999,35,15},{3
0,999,999,999,999,20},{45,40,35,999,999,55},{999,25,15,20,55,999}};
    int ne[6], n, l, j, k, l, t[6][2], min=999,mincost;
    for (i=0;i<TAM;i++)
        for (j=0;j<TAM;j++)
            if (min>cost[i][j]){
                min=cost[i][j];
                k=l;
                l=j;
            }
    mincost=cost[k][l];
    t[0][0]=k;
    t[0][1]=l;
    for (i=0;i<TAM;i++)
        if (cost[i][l]<cost[i][k])
            ne[i]=l;
        else
            ne[i]=k;
    ne[k]=ne[l]=-1;
    for (i=1;i<TAM-1;i++){
        min=999;
        for (k=0;k<TAM;k++){
            if (ne[k]!=-1 && cost[k][ne[k]]<min){
                min=cost[k][ne[k]];
                j=k;
            }
        }
        t[i][0]=j;
        t[i][1]=ne[j];
        mincost=mincost+cost[j][ne[j]];
        ne[j]=-1;
        for (k=0;k<TAM;k++)
            if(ne[k]!=-1 && cost[k][ne[k]]>cost[k][j])
                ne[k]=j;
    }
}

```

```

}
printf("Costo del árbol de expansión mínima=%d\n",mincost);
for (i=0;i<TAM-1;i++)
    printf("Trayecto de %d a %d\n",t[i][0]+1,t[i][1]+1);
getchar();
getchar();
}

```

Programa 9.1

La matriz COSTO del ejemplo anterior junto con una simulación de los valores históricos del vector NEAR se muestra en la tabla 9.1:

	1	2	3	4	5	6	NEAR
1	$\infty$	10	$\infty$	30	45	$\infty$	0
2	10	$\infty$	50	$\infty$	40	25	0
3	$\infty$	50	$\infty$	$\infty$	35	15	2,6
4	30	$\infty$	$\infty$	$\infty$	$\infty$	20	1, 6, 0
5	45	40	35	$\infty$	$\infty$	50	2, 3, 0
6	$\infty$	25	15	20	55	$\infty$	2,0

Tabla 9.1

#### 9.4 La ruta más corta a partir de un origen (Single Source Shortest Paths)

Los grafos pueden ser utilizados para representar carreteras, estructuras de un estado o país con vértices representando ciudades y segmentos que unen los vértices como la carretera. Los segmentos pueden tener asignados pesos que marcan una distancia entre dos ciudades conectadas.

La distancia de un trayecto es definido por la suma del peso de los segmentos. El vértice de inicio se definirá como el origen y el último vértice se definirá como el destino. El problema a considerar será en base a una gráfica dirigida  $G = (V, E)$ , una función de peso  $c(e)$  para los segmentos de  $G$  y un vértice origen  $v_0$ . El problema es determinar el trayecto más corto de  $v_0$  a todos los demás vértices de  $G$ . Se asume que todos los pesos son positivos.

Ejemplo:

Considere el grafo dirigido de la figura 9.4. El número de trayectos también es el número de pesos. Si  $v_0$  es el vértice de origen, entonces el trayecto más corto desde  $v_0$  a  $v_1$  es  $v_0 v_2 v_3 v_1$ . La distancia del trayecto es  $10 + 15 + 20 = 45$ . En este caso, recorrer tres caminos es más económico que recorrer en forma directa  $v_0 v_1$ , el cual tiene un costo de 50.

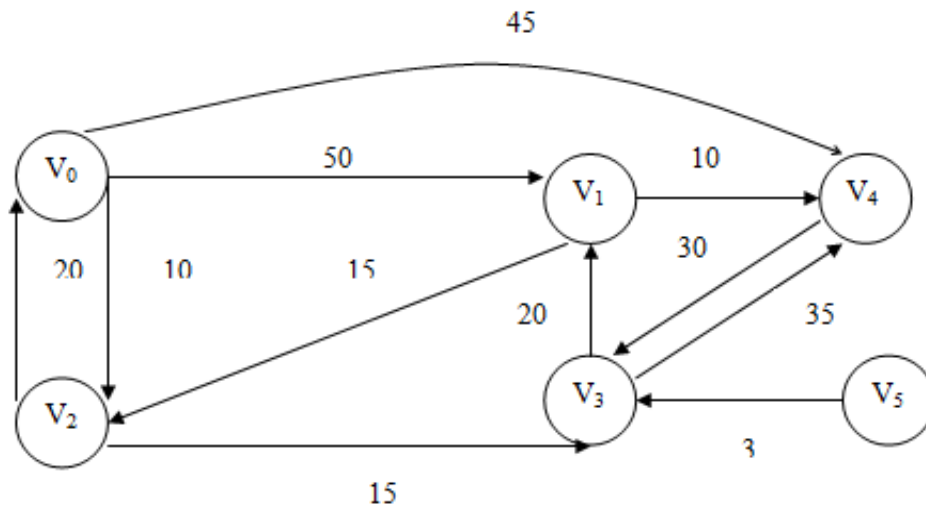


Fig. 9.4 Un ejemplo para el algoritmo del trayecto más corto.

Para formular un algoritmo voraz para generar el trayecto más corto, debemos concebir una solución multietapa. Una posibilidad es construir el trayecto más corto uno por uno. Como una medida de optimización se puede usar la suma de todos los trayectos hasta el momento generados. En orden de que la medida sea mínima, cada trayecto individual debe ser de tamaño mínimo. Si se han construido  $i$  trayectos mínimos, entonces el siguiente trayecto a ser construido debería ser el siguiente trayecto con mínima distancia. El camino greedy o voraz para generar los trayectos cortos desde  $V_0$  a los vértices remanentes es generando los trayectos en orden creciente. Primero, el trayecto más corto al vértice más cercano es generado. Entonces el trayecto más corto al segundo vértice más cercano se genera y así sucesivamente. Para el grafo del ejemplo, el trayecto más cercano para  $V_0$  es  $V_2$  ( $c(V_0, V_2) = 10$ ). Por lo que el trayecto  $V_0 V_2$  será el primer trayecto generado. El segundo trayecto más cercano es  $V_0 V_3$  con una distancia de 25. El trayecto  $V_0 V_2 V_3$  será el siguiente trayecto generado. Para generar los siguientes trayectos se debe determinar i) el siguiente vértice que con el cual deba generar un camino más corto y ii) un camino más corto para éste vértice. Sea  $S$  el conjunto de vértices (incluyendo  $V_0$ ) en el cual el trayecto más corto ha sido generado. Para  $w$  no en  $S$ , sea  $DIST(w)$  la distancia del trayecto más corto desde  $V_0$  yendo sólo a través de estos vértices que están en  $S$  y terminando en  $w$ . Se observa que:

- I. Si el siguiente trayecto más corto es al vértice  $u$ , entonces el trayecto inicia en  $V_0$ , termina en  $u$  y va a través de los vértices localizados en  $S$ .
- II. El destino del siguiente trayecto generado debe de ser aquel vértice  $u$  tal que la mínima distancia,  $DIST(u)$ , sobre todos los vértices no en  $S$ .

- III. Habiendo seleccionado un vértice  $u$  en  $U$  y generado el trayecto más corto de  $V_0$  a  $u$ , el vértice  $u$  viene a ser miembro de  $S$ . En este punto la dimensión del trayecto más corto iniciando en  $V_0$  irá en los vértices localizados en  $S$  y terminando en  $w$  no en  $S$  puede decrecer. Esto es, el valor de la distancia  $DIST(w)$  puede cambiar. Si cambia, entonces se debe a que existe un trayecto más corto iniciando en  $V_0$  posteriormente va a  $u$  y entonces a  $w$ . Los vértices intermedios de  $V_0$  a  $u$  y de  $w$  a  $w$  deben estar todos en  $S$ . Además, el trayecto  $V_0$  a  $u$  debe ser el más corto, de otra forma  $DIST(w)$  no está definido en forma apropiada. También, el trayecto  $u$  a  $w$  puede no contener vértices intermedios.

Las observaciones arriba indicadas forman un algoritmo simple. (El algoritmo fue desarrollado por Dijkstra en 1959). De hecho solo determina la magnitud de la trayectoria del vértice  $V_0$  a todos los vértices en  $G$ .

Se asume que los  $n$  vértices en  $G$  se numeran de 1 a  $n$ . El conjunto se mantiene  $S$  con un arreglo con  $S(i)=0$  si el vértice  $i$  no se encuentra en  $S$  y  $S(i)=1$  si pertenece a  $S$ . Se asume que el grafo se representa por una matriz de costos.

En pseudocódigo el algoritmo y su programación en C se observa en el programa 9.2:

```

Procedure SHORTEST-PATHS( $v$ ,  $COST$ ,  $DIST$ ,  $n$ )
    //  $DIST(j)$  es el conjunto de longitudes del trayecto más corto del vértice  $v$  al
    // vértice  $j$  en la gráfica  $G$  con  $n$  vértices.
    //  $G$  es representada por la matriz de costos  $COST(n, m)$ 
    Boolean  $S(1:n)$ ; real  $COST(1:n, 1:n)$ ,  $DIST(1:n)$ 
    Integer  $u, v, n, num, i, w$ 
    For  $i \leftarrow 1$  to  $n$  do
         $S(i) \leftarrow 0$ ;  $DIST(i) \leftarrow COST(v, i)$ 
    Repeat
         $S(v) \leftarrow 1$   $DIST(v) \leftarrow 0$  // colocar el vértice  $v$  en  $S$ .
        For  $num \leftarrow 2$  to  $n-1$  do // determina  $n - 1$  trayectos desde el vértice  $v$ .
            Escoger  $u$  tal que  $DIST(u) = \min\{DIST(w)\}$ 
             $S(w) = 0$ 
             $S(u) \leftarrow 1$  //Coloca el vértice  $u$  en  $S$ 
            For all  $w$  con  $S(w) = 0$  do
                 $DIST(w) \leftarrow \min(DIST(w), DIST(u) + COST(u, w))$ 
            Repeat
        Repeat
    End SHORTEST-PATH.

```

```

#include<stdio.h>
#include<stdlib.h>
//La ruta más corta a partir de un origen
void generar(int **,int);
void path(int **,int *,int,int);

main(){
    int **cost,*dist,tam,l,j,v;
    printf("***** Ruta mas corta a partir de un origen *****\n");
    printf("\nIntroduce el numero de Vértice142: ");
    scanf("%d",&tam);
    cost=(int **)malloc(sizeof(int *)*tam);
    for(i=0;i<tam;i++)
        cost[i]=(int *)malloc(sizeof(int)*tam);
    dist=(int *)malloc(sizeof(int)*tam);
    for(i=0;i<tam;i++)
        for(j=0;j<tam;j++)
            cost[i][j]=9999;
    generar(cost,tam);
    printf("Introduce el vértice de origen: ");
    scanf("%d",&v);
    printf("La matriz generada es: \n");
    for(i=0;i<tam;i++){
        for(j=0;j<tam;j++)
            printf("%6d",cost[i][j]);
        printf("\n");
    }
    path(cost,dist,tam,v-1);
    printf("\n\nCosto de los caminos\n");
    for(i=0;i<tam;i++)
        printf("Camino %d a %d: %d\n",v,i+1,dist[i]);
    system("PAUSE");
}

void generar(int **cost,int tam){
    int i,nv=0,p,ad;
    printf("Para terminar de introducir un Vértice introduce 99\n");
    while(nv<tam){
        printf("Vertice %d a... \n",nv+1);
        scanf("%d",&ad);
        if(ad==99){
            printf("Termino vertice %d\n",nv+1);
            nv++;
        }
    }
}

```

```

        else if(ad>tam)
            printf("El 143értice no existe \n");
        else{
            printf("Intorduce el peso del Vertice:");
            scanf("%d",&p);
            cost[nv][ad-1]=p;
        }
    }
}

void path(int **cost,int *dist,int tam,int v){
    int u,w,l,num,s[tam],min;
    for(i=0;i<tam;i++){
        s[i]=0;
        dist[i]=cost[v][i];
    }
    s[v]=1;
    dist[v]=0;
    for(num=1;num<tam-1;num++){
        min=9999;
        for(w=0;w<tam;w++){
            if (s[w]==0 && dist[w]<min){
                min=dist[w];
                u=w;
            }
        }
        s[u]=1;
        for(w=0;w<tam;w++){
            if(s[w]==0){
                if(dist[w]<dist[u]+cost[u][w])
                    dist[w]=dist[u]+cost[u][w];
                else
                    dist[w]=dist[u]+cost[u][w];
            }
        }
    }
}

```

Programa 9.2

Un ejemplo se muestra en la figura 9.5.

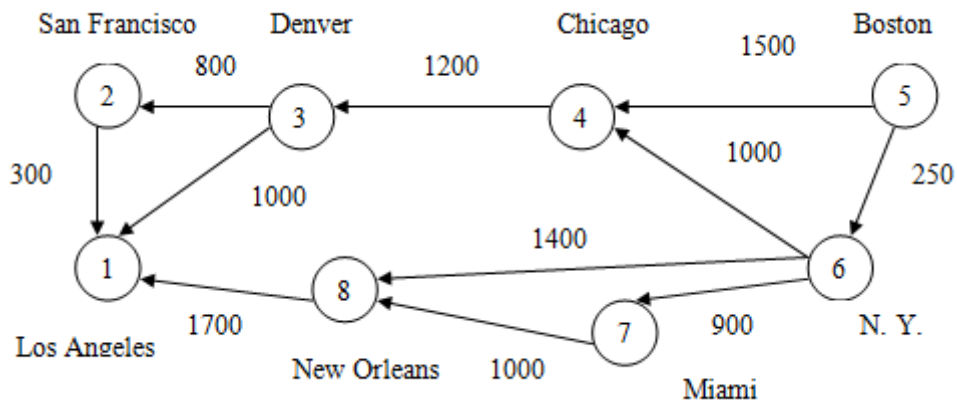


Fig. 9.5 Un ejemplo para el trayecto más corto.

La matriz de costos se muestra en la tabla 9.2:

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0			
6				1000		0	900	1400
7							0	1000
8								0

Tabla 9.2

Si  $v=5$ , nos indica que se busca el trayecto de mínimo costo a todos los nodos desde el nodo 5. Por lo tanto, la corrida se muestra en la tabla 9.3:

Iteración	S	Vértice	1	2	3	4	5	6	7	8
Inicial	-									
1	5	6	$\infty$	$\infty$	$\infty$	1500	$\infty$	250	$\infty$	$\infty$
2	5, 6	7	$\infty$	$\infty$	$\infty$	1250	0	250	1150	1650
3	5, 6, 7	4	$\infty$	$\infty$	$\infty$	1250	0	250	1150	1650
4	5, 6, 7, 4	8	$\infty$	$\infty$	2450	1250	0	250	1150	1650
5	5, 6, 7, 4, 8	3	3350	$\infty$	2450	1250	0	250	1150	1650
6	5, 6, 7, 4, 8, 3	2	3350	3250	2450	1250	0	250	1150	1650
	5, 6, 7, 4, 8, 3, 2		3350	3250	2450	1250	0	250	1150	1650

Tabla 9.3



### Ejercicios.

1. Dada la siguiente matriz de costos:

	$\infty$	10	$\infty$	40	30	$\infty$
	10	$\infty$	50	$\infty$	40	45
$C = \infty$	50	$\infty$	$\infty$	40	10	
	30	$\infty$	$\infty$	$\infty$	$\infty$	40
	50	40	50	$\infty$	$\infty$	10
	$\infty$	30	20	40	50	$\infty$

Obtener el árbol de expansión mínima.

2. Dada la siguiente matriz de costos:

	1	2	3	4	5	6	7	8
1	0							
2	400	0						
3	800	600	0					
4			1200	0				
5				1400	0	250		
6				1200		0	1000	1600
7							0	1000
8								0

Obtener el trayecto más corto de 5 a 1

## Bibliografía

1. Backman, k. (2012). *Structured Programming with C++*. Bookboom.com.
2. Cairó/Gardati. (2000). *Estructura de Datos*. México: Mc Graw hill.
3. Ceballos, J. (2015). *C/C++ Curso de Programación*. México: AlfaOmega Ra-Ma.
4. Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms*. New York: Mc Graw Hill.
5. Deitel, P., & Deitel, H. M. (1995). *Como programar en C/C++*. México: Prentice Hall.
6. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (1991). *Fundamentals of Software Engineering*. New Jersey: Prentice Hall.
7. Horowits, E. (1978). *Fundamentals of Computer Algorithms*. New York: Computer Science Press.
8. Levin, G. (2004). *Computación y Programación Moderna, perspectiva integral de la Informática*. México: Addison Wesley.
9. Loomis, M. E. (2013). *Estructura de Datos y Organización de Archivos*. México: Prentice Hall.
10. *Serie de Fibonacci*. (s.f.). Recuperado el 2013 de 07 de 13, de <http://www.ugr.es/~eaznar/fibo.htm>